MR-PDP: Multiple-Replica Provable Data Possession

Reza Curtmola
Department of Computer Science
Purdue University
crix@cs.purdue.edu

Osama Khan, Randal Burns, Giuseppe Ateniese
Department of Computer Science
Johns Hopkins University
{okhan, randal, ateniese}@cs.jhu.edu

Abstract

*Many storage systems rely on replication to increase the availability and durability of data on untrusted storage systems. At present, such storage systems provide no strong evidence that multiple copies of the data are actually stored. Storage servers can collude to make it look like they are storing many copies of the data, whereas in reality they only store a single copy. We address this shortcoming through multiple-replica provable data possession (*MR-PDP*): A provably-secure scheme that allows a client that stores t replicas of a file in a storage system to verify through a challenge-response protocol that (1) each unique replica can be produced at the time of the challenge and that (2) the storage system uses t times the storage required to store a single replica.* MR-PDP *extends previous work on data possession proofs for a single copy of a file in a client/server storage system [4]. Using* MR-PDP *to store t replicas is computationally much more efficient than using a single-replica PDP scheme to store t separate, unrelated files (*e.g.*, by encrypting each file separately prior to storing it). Another advantage of* MR-PDP *is that it can generate further replicas on demand, at little expense, when some of the existing replicas fail.*

1 Introduction

We address the problem of creating multiple unique replicas of a file in a distributed storage system. This allows a client to query the distributed system to ensure that there are multiple unique copies of its file stored in the network even when storage sites collude. Our original motivation was to give a data owner that archives data with third-party storage services, such as Amazon S3 [3] or the Storage Request Broker [26], the ability to perform introspection and maintenance on its data. However, these techniques apply to all replication-based, distributed, and untrusted storage systems, including peer-to-peer storage systems [1, 11, 18, 19, 21].

Replication is a fundamental principle in ensuring the availability and durability of data [16]. Managing the number and placement of replicas is critical to this process. Systems re-replicate data when replicas fail [10, 12], evaluate the correctness of replicas in the system [20], and move replicas among sites to meet availability goals [2, 9].

However, distributed storage systems lack constructs that allow them to securely determine the number and location of replicas in the system. Distributed storage systems that perform replica maintenance often have storage sites cross-check the contents of replicas through content hashing [10, 20]. Recently, there has been much interest in having clients (that do not have a copy of the data) check that servers have a copy of the data [4, 17, 23, 24]. Both types of protocols are vulnerable to collusion attacks in which multiple servers that appear to be storing multiple replicas are in fact storing only a single copy of the data. In general, this can be done by redirecting and forwarding challenges from the multiple sites to the single site that stores the data.

Storing a single copy, while appearing to store many copies, benefits servers; redirection and forwarding attacks are practical and servers are motivated to perform them. In peer-to-peer storage systems, servers can use this attack to freeload [15], using resources in the system without contributing a commensurate amount of resources back to peers. Third-party, outsourced storages sites can use this type of attack to sell the same storage space multiple times. In both cases, clients (data owners) remain unaware of the reduction in the availability and durability of data that results from the loss of replicas.

The natural solutions to this problem result in significant time, space, and management overheads. A simple way to make replicas unique and differentiable is by encryption. If the client were to generate each replica by encrypting the data under different keys that are kept secret from the servers, then the servers could not compare the replicas, use one replica to answer challenges for another, or compress replicas with respect to one another. Each replica is a separate file to be created and checked individually, using a protocol for checking data possession. (Cross-checking proto-

cols [10,20] would not work as encrypted replicas cannot be compared.) In this simple approach, the computation time for both replica creation and checking grow linearly in the number of replicas.

In this paper, we describe cryptographic constructs that allow for a client to securely establish that the network stores multiple unique replicas. The scheme uses a constant amount of metadata for any number of replicas and new replicas may be created dynamically without pre-processing the file again. Also, multiple replicas may be checked concurrently, so that checking t replicas is less expensive than t times the cost of checking a single replica. Thus, our solution overcomes the time, space, and management overheads associated with the natural solution mentioned previously.

These constructs build upon the provable data possession (PDP) [4] client/server data checking scheme and, thus, inherit PDP's benefits. PDP allows a client to store a file on a server so that it may later challenge the server to prove possession. In responding to the challenge, the server provides a probabilistic proof that it has access to the exact data stored by the client previously. Because the challenge is probabilistic, it is I/O efficient; the server accesses a small constant amount of data in generating the proof. The client stores only a small $\mathrm{O}(1)$ amount of key material to verify the server's proof. The scheme introduced the notion of *homomorphic verification tags*, which are crucial for achieving low-bandwidth verification. A set of verification tags is stored on the server together with each file (one tag per file block), allowing the client to check possession of file blocks without access to the actual blocks; moreover, these tags can be aggregated, resulting in compact proofs of possession. As a result, the scheme uses $\mathrm{O}(1)$ bandwidth: The challenge and the response are each approximately 200 bytes. Thus, PDP allows a client to verify data possession without retrieving the data from the server and without having the server access the entire file; this makes it practical to check possession of large amounts of data that are stored remotely.

We extend PDP to apply to multiple replicas so that a client that initially stores t replicas can later receive a guarantee that the storage system can produce t replicas, each of which can be used to reconstruct the original file data. A replica comprises the original file data masked with randomness generated by a pseudo-random function (PRF). As each replica uses a different PRF, replicas cannot be compared or compressed with respect to each other. We modify the homomorphic verification tags of PDP [4] so that a single set of tags can be used to verify any number of replicas. These tags need to be generated a single time against the original file data. Thus, replica creation is efficient and incremental; it consists of unmasking an existing replica and re-masking it with new randomness. In fact, our multiple-replica PDP scheme (MR-PDP) is almost as efficient as a single-replica PDP scheme in all the relevant parameters.

Establishing the existence of different replicas is critical to making PDP usable in distributed systems. It removes a server's ability to cheat in order to reduce storage requirements. At the same time, it preserves the provable security and I/O and network efficiency of PDP when applied to complex distributed systems.

2 The Multiple-Replica PDP Guarantees

In this section, we present the guarantees we seek to achieve with our multiple-replica PDP system. We also discuss the practical implications of these guarantees and some of their limitations.

Our multiple-replica PDP system uses replication in order to improve the data availability and reliability of a single-replica PDP system. The basic principle of data replication is well understood. By storing redundant copies of the data, one can ensure that if some of the copies are destroyed, the data can still be recovered from the remaining copies.

Most replication systems can tolerate failures if the failure modes of the replicas are independent. In the absence of failure independence, all replicas may fail simultaneously and replication does not help (*e.g.*, because all replicas are stored in the same geographical location or because data dependencies exist among replicas). When the storage servers are non-malicious, geographic diversity can ensure failure independence; *e.g.*, the client stores data on servers that have different locations. Thus, the main function of the replication system is to tolerate independent, accidental (*i.e.*, non-malicious) failures, such as hardware failures.

The situation is different when storage servers are untrusted, *i.e.*, servers are malicious and can collude. A replication system relying on untrusted servers cannot offer the same assurance level as a system relying on benign servers, because failure independence cannot be assumed. Even if the client initially stores replicas on servers in different geographic locations, the servers can then move all the replicas to one location and access them from that location on demand. Such a system is not more reliable than a single-replica system, even though it leads the client to believe so. We are not aware of any system that guarantees improved reliability against such a strong adversarial model, and our system does not offer it either. Establishing the physical location of the data is an important open problem.

Replication systems that rely on untrusted servers have another generic limitation. To prove data availability, the servers can produce replicas on demand upon a client's challenge; however, this does not prove that the actual replicas are stored at all times. For example, malicious servers may chose to introduce dependencies among replicas, by encrypting the replicas before storing them. Replicas can then be decrypted and served on demand whenever they are

requested by clients. By storing the encryption key in a single location, the malicious servers can effectively negate any reliability improvements achieved by storing the replicas at different locations. Loss of the encryption key means loss of all the replicas.

Given these generic limitations of replication systems that rely on fully dishonest servers, we consider a model in which storage servers are rational and economically motivated. In this context, cheating is meaningful only if it cannot be detected and if it achieves some economic benefit (*e.g.*, using less storage than required by the contract). We note that such an adversarial model is reasonable and captures many practical settings in which malicious servers will not cheat and risk their reputation, unless they can achieve a clear financial gain.

A client that stores equal-sized replicas $\mathtt{F}_1, \ldots, \mathtt{F}_t$ of a file $\mathtt{F}$ with a multiple-replica PDP system achieves the following *multiple-replica PDP guarantees*:

– **MRG1**: When it receives a challenge from the client, the system can produce replicas $\mathtt{F}_1, \ldots, \mathtt{F}_t$ such that the client can recover the original file from any of these replicas;

– **MRG2**: The amount of storage used by the system is at least t times the storage required by the system to store one replica.

The practical implications of these guarantees are that the multiple-replica PDP system can produce on demand the data of the different replicas and that it cannot eliminate replicas to save on storage space. The most practical and beneficial attacks are those that reduce the servers resource consumption. **MRG2** ensures that servers cannot cheat by claiming to store t replicas (and charging for storage of t replicas), but storing fewer than t replicas; servers are also motivated to store the actual replicas, because introducing data dependencies does not achieve any storage savings and, thus, it does not provide any economic benefit. Even with **MRG2** in place, servers may still try to cheat by deleting parts of the data that are rarely accessed and hoping this will go undetected; **MRG1** addresses this by ensuring that, with high probability, servers possess replicas in their entirety.

3 Multiple-Replica Provable Data Possession (MR-PDP) Schemes

The client C has a file $\mathtt{F}$, viewed as a finite ordered collection of n blocks: $\mathtt{F} = (\mathtt{f}_1, \mathtt{f}_2, \ldots, \mathtt{f}_n)$, in which every file block has β bits. The client wants to generate and store t replicas of the file, $\mathtt{F}_1, \ldots, \mathtt{F}_t$, on t servers, $S_1, \ldots, S_t$.

3.1 Definition of a MR-PDP system

We follow the definition of a PDP system introduced in the context of storing a single replica at a single server [4]

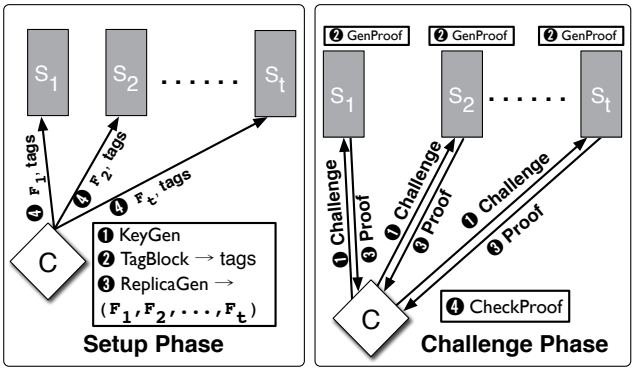

Figure 1. The Setup **and** Challenge **phases of a multiple-replica PDP system. During** Setup**, the client C generates t replicas and tags and stores them on t servers. During** Challenge**, for a** *complete challenge***, the client executes t** *individual challenges***.**

and adapt this definition to the setting of multiple-replica PDP in which multiple servers store multiple replicas.

A multiple-replica PDP scheme consists of five algorithms (KeyGen, ReplicaGen, TagBlock, GenProof, CheckProof). KeyGen is a key generation algorithm that is run by the client to setup the scheme. ReplicaGen is used by the client to generate a replica of $\mathtt{F}$. TagBlock is run by the client to generate the verification tag for a file block. GenProof is run by a server and CheckProof is run by the client in order to generate and to validate a proof of data possession respectively.

As shown in Fig. 1, a multiple-replica PDP system has three phases: Setup, Challenge and Replicate.

– Setup: The client initializes the system by executing KeyGen, uses ReplicaGen to generate t replicas of $\mathtt{F}$ and pre-processes the file by using TagBlock to generate the verification tags for these replicas. The client then stores the replicas and the verification tags at the servers and retains a small, constant amount of information that will be used in the Challenge phase. Finally, the client may delete the file, the replicas, and the tags from its local storage.

– Challenge: The client can execute *individual challenges* or *complete challenges*. For individual challenges, the client interacts with a particular server S_u and determines if S_u possesses replica $\mathtt{F}_u$ at the time of the challenge. A complete challenge consists of t individual challenges, which can be executed in parallel: The client challenges server S_u to prove possession of replica $\mathtt{F}_u$, for $1 \leq u \leq t$. There is no restriction on how many times the client may run the challenge protocol to ascertain if servers still possess file replicas.

– Replicate: This phase allows the client to perform replica maintenance: The client can dynamically create new replicas using ReplicaGen in order to maintain a desired repli-

cation factor whenever it detects a replica failure (*i.e.*, some server cannot prove possession of a replica).

3.2 Multiple-Replica PDP Schemes

In this section, we present our main result, an efficient multiple-replica PDP scheme (MR-PDP) that guarantees that the storage servers are storing multiple unique replicas. To illustrate the pitfalls behind building such a scheme, we first present two warmup schemes. The first one does not ensure the multiple-replica PDP guarantees and the second one is not as efficient as we would like. Our main scheme overcomes these drawbacks. It successfully enforces the multiple-replica PDP guarantees and its efficiency is close to that of a single-replica PDP scheme.

A first attempt. Given a PDP scheme that ensures possession of a single file replica [4], it is tempting to use this scheme repeatedly t times to ensure possession of t file replicas at t servers: The client generates t different sets of tags using different key material and stores a copy of the file $\mathtt{F}$ and a set of tags at each server S_u, for $1 \leq u \leq t$; the client then uses the single-replica PDP scheme to challenge the servers in proving possession of the replicas. Unfortunately, this scheme succumbs to the following collusion attack, which breaks the **MRG2** guarantee: Because the replicas of the file are identical, the servers can collude and collectively store only one replica, instead of storing t replicas, unbeknownst to the client.

A second attempt (ENC-PDP). To prevent the collusion attack described above, the client needs to generate and store unique and differentiable file replicas.

We present a generic transformation that allows the client to transform any PDP scheme that ensures possession of a single file replica [4, 17] into a scheme that allows the client to create and store t unique and differentiable replicas at t servers: The client creates t different replicas by encrypting the original file under t different keys, stores these t replicas and then uses the single-replica PDP scheme to enforce possession of each of the t replicas. In essence, this is equivalent to the client applying the single-replica PDP scheme independently on t different files. We refer to such a PDP scheme as ENC-PDP. While this transformation is generic, the efficiency of the resulting ENC-PDP scheme is not optimal: The client cost (for both the Setup and Challenge phases) is t times larger than the client cost for the single-replica PDP scheme. It was shown in previous work [4] that the cost of the pre-processing phase represents the limiting factor for PDP schemes, as perceived by the client. Thus, applying a single-replica PDP scheme independently t times will require a significant effort on the client and may render the scheme impractical, especially for large values of t. Moreover, in order to create a new replica during the Replicate phase, the client has to perform the same amount of computation that was necessary to generate one of the original t replicas.

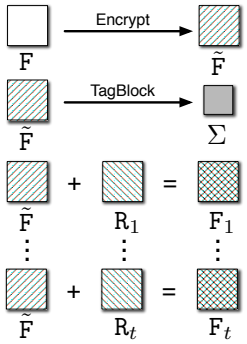

Figure 2. During Setup the client encrypts the file $\mathtt{F}$ into $\tilde{\mathtt{F}}$ and uses $\tilde{\mathtt{F}}$ to generate Σ (the set of verification tags) and t different file replicas $\mathtt{F}_1, \ldots, \mathtt{F}_t$.

An efficient solution (MR-PDP). Our goal is to create t unique and differentiable replicas suitable for use in PDP based on pre-processing the input file "a single time." A single time means that the cost of the pre-processing phase scales as $\mathrm{O}(n)$, the number of file blocks, rather than the $\mathrm{O}(nt)$ required when pre-processing each replica separately as in ENC-PDP. Another goal is to have a "cheap" way to dynamically generate new replicas; in other words, generating new replicas should be able to reuse the effort put in generating the first replica. Next, we give an overview of our main scheme, MR-PDP, followed by its detailed description. In Section 4, we give a simple step-by-step example of how MR-PDP operates.

Overview of the MR-PDP scheme. In the Setup phase, the client pre-processes the file to be stored. As shown in Fig. 2, the client first encrypts the original file $\mathtt{F}$ into $\tilde{\mathtt{F}}$, and then uses $\tilde{\mathtt{F}}$ to generate a set of verification tags (one tag per file block). The client uses the encrypted file $\tilde{\mathtt{F}}$ to also generate t different file replicas, where each replica $\mathtt{F}_u$ is obtained by masking the blocks of $\tilde{\mathtt{F}}$ with a random value $\mathtt{R}_u$ (specifically generated for that replica). The client then stores on each server S_u a replica $\mathtt{F}_u$ and the set of verification tags. Note that the client generates a single set of verification tags, independently of the number of replicas created initially during Setup or later during Replicate.

In the Challenge phase, the client challenges S_u to prove possession of a subset of blocks from replica $\mathtt{F}_u$ (Fig. 3). By sampling a random subset of blocks in each challenge, the client ensures that (a) the server cannot reuse answers to previous challenges, (b) the server's overhead is bounded by the number of sampled blocks (usually a small number), and (c) the data possession guarantee holds over the entire replica $\mathtt{F}_u$. Server S_u computes a proof of possession based on the client's challenge, the stored replica $\mathtt{F}_u$, and the set

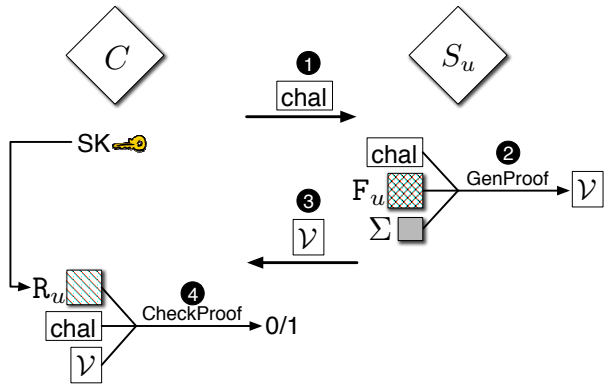

Figure 3. In the Challenge phase, an individual challenge for replica $\mathtt{F}_u$ consists of a 4-step protocol between the client C and server S_u: (1) C challenges S_u to prove possession of replica $\mathtt{F}_u$; (2) S_u generates a proof of possession $\mathcal{V}$; (3) S_u sends proof $\mathcal{V}$ to C; (4) C checks the validity of $\mathcal{V}$.

of verification tags. The client checks the validity of the proof received from S_u based on the random value $\mathtt{R}_u$ (recomputed using its secret key), the challenge, and the proof of possession.

In the Replicate phase, the client creates a new replica $\mathtt{F}_j$ in the same fashion that replicas were created in the Setup phase: It masks the encrypted file $\tilde{\mathtt{F}}$ with random value $\mathtt{R}_j$.

The most expensive operation for the client is the generation of the verification tags, but this is done only once during Setup. New replicas are tied to the same set of verification tags generated during Setup. Thus, generating a new replica is a lightweight operation, because it does not require any expensive exponentiations on the client. Note that the Challenge and Replicate phases can alternate.

Details of the MR-PDP scheme. We start by introducing some additional notation used by the construction, followed by a detailed description of the Setup, Challenge and Replicate phases of MR-PDP.

Preliminaries. We write $f_k(x)$ to denote f keyed with key k applied on input x. Let $p = 2p'+1$ and $q = 2q'+1$ be safe primes and let $N = pq$ be an RSA modulus. Let g be a generator[1] of QR_N, the unique cyclic subgroup of $\mathbb{Z}_N^*$ of order $p'q'$ (*i.e.*, QR_N is the set of quadratic residues modulo N). All exponentiations are performed modulo N, which we often omit writing explicitly. Let $h : \{0,1\}^* \rightarrow QR_N$ be a secure deterministic hash-and-encode function[2] that maps strings uniformly to QR_N. Let $\kappa, \ell, \lambda, \epsilon$ be security parameters and let H be a cryptographic hash function. In addition, we make use of a pseudo-random function ψ and two pseudo-random permutations π, E_K defined as:

- $\psi : \{0,1\}^\kappa \times \{0,1\}^{\log_2(t)+log_2(n)} \rightarrow \{0,1\}^{\beta+\epsilon}$;
- $\pi : \{0,1\}^\kappa \times \{0,1\}^{\log_2(n)} \rightarrow \{0,1\}^{\log_2(n)}$;
- $E_K : \{0,1\}^\kappa \times \{0,1\}^\beta \rightarrow \{0,1\}^\beta$,

where β is the bit size of a file block, and both β and ϵ must be large enough (*e.g.*, $\beta \geq 160, \epsilon \geq 80$). In practice, E_K can be implemented with a standard symmetric cipher.

MR-PDP: Setup **phase.** The client runs KeyGen to generate the system's public and secret parameters. The public parameters are N and g. The secret parameters are e, d, v, z and K. The values e and d are chosen such that e is a large secret prime, both are larger than λ, and $ed \equiv 1 \bmod p'q'$. The parameters v, z, K are κ-bit randomly chosen values that will be used as key material.

The client first obtains an encrypted file $\tilde{\mathtt{F}}$ by encrypting the original file $\mathtt{F}$ with the key K: $\tilde{\mathtt{F}} = \{\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n\}$, where $\mathtt{b}_i = E_K(\mathtt{f}_i)$, for $1 \leq i \leq n$. The client then generates t different file replicas as follows. Replica $\mathtt{F}_u$, for $1 \leq u \leq t$, is generated by masking the blocks of the encrypted file $\tilde{\mathtt{F}}$ with random values computed specifically for the u-th replica. More precisely, $\mathtt{F}_u = (\mathtt{m}_{u,1}, \ldots, \mathtt{m}_{u,n})$, where $\mathtt{m}_{u,i} = \mathtt{b}_i + \mathtt{r}_{u,i}$, for $1 \leq i \leq n$. We emphasize that the blocks $\mathtt{b}_i$ and $\mathtt{r}_{u,i}$ are added as large integers in $\mathbb{Z}$. Note that the blocks $\mathtt{b}_i$ could be arbitrarily large (even larger than N). The random-looking value $\mathtt{r}_{u,i}$ is obtained as the output of the pseudo-random function ψ applied on $u||i$ (*i.e.*, $\mathtt{r}_{u,i} = \psi_z(u||i)$).

The client also generates the set of verification tags $\Sigma = \{\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n\}$ by computing a tag $\mathtt{T}_i$ for each block $\mathtt{b}_i$ of the encrypted file $\tilde{\mathtt{F}}$: $\mathtt{T}_i = (h_i \cdot g^{\mathtt{b}_i})^d \bmod N$, for $1 \leq i \leq n$, where $h_i = h(v||i)$. Note that each tag $\mathtt{T}_i$ is a function of the tagged block $\mathtt{b}_i$, which will allow a server S_u to prove possession of the corresponding block $\mathtt{m}_{u,i}$ from replica $\mathtt{F}_u$. Also, note that each tag $\mathtt{T}_i$ includes information about the index i, which binds the tag to the i-th block and prevents a server from using the tag to prove possession for a block with a different index. The set of verification tags is computed only once and all the replicas generated during the Setup and Replicate phases will be tied to this set of tags.

At the end of the Setup phase, the client stores on each server S_u the corresponding replica $\mathtt{F}_u$ and the set of verification tags Σ. The client may then delete the original file, the replicas and the verification tags from its local storage. Note that while the servers need to collectively store t distinct replicas, they only have to collectively store one copy of Σ, which may lead to a reduction in the additional server storage when compared to the ENC-PDP scheme.

MR-PDP: Challenge **phase.** To check possession of replica $\mathtt{F}_u = (\mathtt{m}_{u,1}, \ldots, \mathtt{m}_{u,n})$, the client challenges server S_u to

[1] g is obtained as $g = a^2$, where $a \overset{R}{\leftarrow} \mathbb{Z}_N^*$ and $\gcd(a \pm 1, N) = 1$.

[2] h is computed by squaring the output of the full-domain hash function for the provably secure FDH signature scheme [6, 7] based on RSA. We refer the reader to [6] for ways to construct an FDH function out of regular hash functions, such as SHA-1. Alternatively, h can be the deterministic encoding function used in RSA-PSS[8]

prove possession of a random subset of blocks from $\mathtt{F}_u$. Previous work [4] showed that if the server is missing a fraction of the data, then the number of blocks that needs to be checked in order to detect server misbehavior is in the order of $O(1)$. For example, if the server S_u is missing 1% of the replica $\mathtt{F}_u$, then the client only needs to ask proof of possession for $c = 460$ randomly chosen blocks of $\mathtt{F}_u$.

The client's challenge for S_u includes c (the number of challenged blocks), k (the key for the pseudo-random function π which determines the challenged blocks) and $g_s = g^s \bmod N$. Here k is a κ-bit value, $s \in \mathbb{Z}_N^*$, and both k and s are chosen randomly for each challenge.

Upon receipt of a challenge, server S_u determines the indexes of the challenged blocks $i_1, \ldots, i_c$ from replica $\mathtt{F}_u$ using the pseudo-random permutation π keyed with k: $i_j = \pi_k(j)$, for $1 \leq j \leq c$. S_u then generates a proof of possession $\mathcal{V} = (\mathtt{T}, \rho)$ for these blocks and returns it to the client. The first value of the proof, $\mathtt{T}$, is computed by multiplying the verification tags corresponding to the challenged blocks: $\mathtt{T} = \mathtt{T}_{i_1} \cdot \ldots \cdot \mathtt{T}_{i_c}$. The second value, ρ, is computed by raising g_s to the sum of the challenged replica blocks: $\rho = g_s^{\mathtt{m}_{u,i_1} + \ldots + \mathtt{m}_{u,i_c}} \bmod N$.

Let $\mathtt{b}_{\mathsf{chal}} = \sum_{j=1}^{c} \mathtt{b}_{i_j}$, $\mathtt{m}_{\mathsf{chal}} = \sum_{j=1}^{c} \mathtt{m}_{u,i_j}$ and $\mathtt{r}_{\mathsf{chal}} = \sum_{j=1}^{c} \mathtt{r}_{i_j}$, be the sum of the challenged blocks in the encrypted file $\tilde{\mathtt{F}}$, the sum of the challenged blocks in the replica $\mathtt{F}_u$, and the sum of the random values used to obtain the blocks in $\mathtt{F}_u$ from the blocks of $\tilde{\mathtt{F}}$ respectively. From the definition of the replica blocks, we have $\mathtt{m}_{\mathsf{chal}} = \mathtt{b}_{\mathsf{chal}} + \mathtt{r}_{\mathsf{chal}}$.

If the proof of possession $\mathcal{V} = (\mathtt{T}, \rho)$ that the client gets back has been correctly computed, then $\mathtt{T}$ and ρ should look as follows: $\mathtt{T} = (h_{i_1} \cdot \ldots \cdot h_{i_c} \cdot g^{\mathtt{b}_{\mathsf{chal}}})^d$, $\rho = g_s^{\mathtt{m}_{\mathsf{chal}}}$. The client verifies the validity of the proof by checking if a relation between $\mathtt{T}$ and ρ holds, namely if $(\frac{\mathtt{T}^e}{h_{i_1} \cdot \ldots \cdot h_{i_c}} \cdot g^{\mathtt{r}_{\mathsf{chal}}})^s$ equals ρ. The correctness of this check results from the commutativity of the exponents in $(g^{\mathtt{m}_{\mathsf{chal}}})^s$ and $(g^s)^{\mathtt{m}_{\mathsf{chal}}}$. The intuition behind this check is that the client can "re-generate" the random values $\mathtt{r}_{u,i}$ (used to generate replica $\mathtt{F}_u$ from the encrypted file $\tilde{\mathtt{F}}$) and then combine them with the blocks of the encrypted file $\tilde{\mathtt{F}}$ in the exponent (these are contained in the value $\mathtt{T}$ in the proof), so that it can verify if they match the replica blocks (used by the server to compute the value ρ in the proof). The algebraic properties of our construct allow the client to perform this check without retrieving the replica blocks from the server.

MR-PDP: Replicate phase. The client can dynamically generate a new replica $\mathtt{F}_u$ from the encrypted file $\tilde{\mathtt{F}}$. If it does not have $\tilde{\mathtt{F}}$ in its local storage, the client retrieves any of the existing replicas, say $\mathtt{F}_w$, and unmasks it in order to recover $\tilde{\mathtt{F}}$ (*e.g.*, $\mathtt{b}_i = \mathtt{m}_{w,i} - \mathtt{r}_{w,i}$, for $1 \leq i \leq n$). The new replica $\mathtt{F}_u$ is derived from $\tilde{\mathtt{F}}$ by using the same masking method that was used to derive replicas during the Setup phase. The replica creation process is lightweight, because it does not require any expensive exponentiations on the client. This allows the client to easily create new replicas on demand, meeting an essential requirement of any replica management system.

Efficiency of the MR-PDP scheme. Our multiple-replica PDP scheme is as efficient as a single-server PDP scheme in most of the parameters. We express the computation cost as the cost of performing modular exponentiations. Preprocessing in the Setup phase requires $O(n)$ computation on the client and is independent on the number of replicas. An individual challenge in the Challenge phase requires $O(1)$ computation for both the client and the challenged server (in fact, each has to perform roughly 2 exponentiations). Also, a server only needs to access $O(1)$ blocks to answer an individual challenge. The communication cost for an individual challenge is also $O(1)$, because the client's challenge and the server's reply each have around 200 bytes. The client stores only a small, constant amount of key material; the storage servers need to store a single set of verification tags (in addition to the actual replicas), regardless of the number of replicas. The client can cheaply generate a new replica during the Replicate phase, because this does not involve any exponentiations.

Security of the MR-PDP scheme. The security of the MR-PDP scheme is captured by the following theorem:

Theorem 3.1 MR-PDP *achieves the multiple-replica PDP guarantees* **MRG1** *and* **MRG2**.
The proof, omitted due to space limitations, follows from the security proof for a single-replica PDP scheme [4].

Remarks. Instead of sending the value ρ, the server could hash this value and send $H(\rho)$ (the client's check would change accordingly). For ease of exposition, we presented MR-PDP without this optimization that would further reduce the size of a proof of possession.

Also, note that the MR-PDP scheme achieves *private verifiability* (only the client, owner of the data, can check data possession), while single-replica PDP schemes [4] also achieve *public verifiability*.

MR-PDP proves possession of the sum of the challenged blocks $\mathtt{m}_{u,i_1} + \ldots + \mathtt{m}_{u,i_c}$, which offers a level of security adequate for most practical purposes and leads to very efficient protocols. The client can be assured that the S_u possesses each one of the challenged blocks by asking proof for $a_{i_1} \cdot \mathtt{m}_{u,i_1} + \ldots + a_{i_c} \cdot \mathtt{m}_{u,i_c}$, where the coefficients $a_{i_1}, \ldots, a_{i_c}$ are randomly chosen for each challenge. However, this stronger guarantee requires additional computation at both server and client (see [4] for more details).

4 A MR-PDP Example

Fig. 4 demonstrates the operation of MR-PDP when the client performs an individual challenge for replica $\mathtt{F}_u$ by

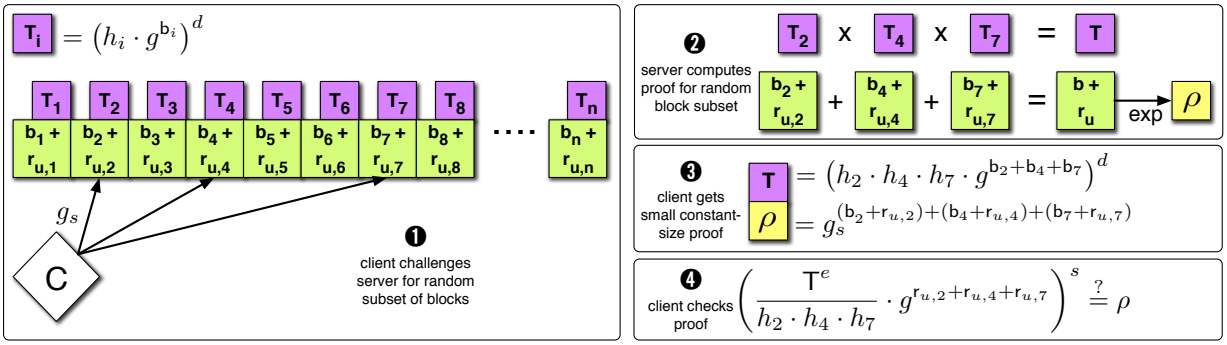

Figure 4. In the Challenge **phase, an individual challenge for replica $\mathtt{F}_u$ consists of a 4-step protocol between the client C and server S_u: (1) C challenges S_u to prove possession of replica blocks with indices 2, 4 and 7; (2) S_u generates a proof of possession $\mathcal{V} = (\mathtt{T}, \rho)$ for the challenged blocks; (3) S_u sends proof $\mathcal{V}$ to C; (4) C checks the validity of $\mathcal{V}$.**

asking server S_u for proof of possession of replica blocks 2, 4, and 7. It provides an informal run-through of the protocol that gives insight into the operation of MR-PDP.

To respond to a challenge successfully, the server must have available the replica $\mathtt{F}_u$, comprised of blocks $(\mathtt{b}_1 + \mathtt{r}_{u,1}), \ldots, (\mathtt{b}_n + \mathtt{r}_{u,n})$, and the verification tags, which are of the form $(h_i \cdot g^{\mathtt{b}_i})^d$ for block $\mathtt{b}_i$. This replica differs from other replicas in the system only in the masking information, *i.e.*, $\mathtt{r}_{u,i}$ for the i-th block. Informally, each tag consists of (1) a secure hash of the block index $h(i)$, which binds the tag to the block offset, and (2) the block's data exponentiated, which binds the tag to the data. Both parts are bound to the user's private key d.

The client submits a random challenge to the server, which consists of a new base of exponentiation $g_s = g^s \bmod N$, in which s is drawn at random and g is the base used in the verification tags.

The client also selects a random subset of blocks, which the server must possess in order to pass the challenge (step 1). In doing so, the client achieves a probabilistic possession guarantee by sampling (spot checking) the server. This random selection is part of the PDP scheme and limits the I/O overhead at the server.

In response to this challenge, the server computes the proof of possession that consists of two values: $\mathtt{T}$ and ρ (step 2). The first value, $\mathtt{T}$, is assembled by multiplying the tags of the challenged blocks. The second value, ρ, is derived from the contents of the challenged blocks themselves. The server sums the file blocks (seen as large numbers) and then exponentiates to this sum. By summing first, the server calculates the proof with a single exponentiation, regardless of how many blocks are challenged. The size of the data used for exponentiation is $\beta \log c$ when challenging c blocks of size β.

The proof of possession is then transferred to the client (step 3). It is of small constant size, because all computations are performed modulo N. We use 1024-bit RSA moduli in our implementation, which makes the size of data transferred around 400 bytes. The client verifies the proof based on its knowledge of the masking information $\mathtt{r}_{u,i}$ and its selection of the random challenge s (step 4). The masking information can be generated on demand and, thus, has not been stored by the client.

The protocol verifies both the specific replica in addition to ensuring that the replica corresponds to the original file. The verification tags have no replica information in them and do not include any masking data. Thus, they verify the original file. The same set of verification tags applies to all replicas. But the client uses the replica-specific masking information to ensure that the data came from the specific checked replica.

5 Experimental Evaluation

We evaluate MR-PDP against the ENC-PDP solution presented in Section 3. ENC-PDP is the natural, obvious solution to prevent collusion among storage servers. It creates unique, incomparable replicas by encrypting files under different keys. It then stores the replicas with an untrusted storage system and uses standard, single-replica PDP [4] to check that servers possess data. The combination of encryption and single-replica PDP allows ENC-PDP to meet the multiple-replica guarantees put forth in Section 2.

Our performance evaluation focuses on the key performance metrics of PDP and MR-PDP. Pre-processing performance (the time to generate tags and, for MR-PDP, to mask the file) governs the write (creation) throughput of the system [4]. We also look at server performance during the challenge phase. Critical to the viability of any data checking system will be that it does not burden storage servers with I/O or computation. This is PDP's principal benefit [4]. Less important is computation at the client during the challenge phase, although we evaluate this as well.

All experiments were conducted on a PowerPC G5 system with two CPUs running at 2 GHz each, and with a 512 KB cache, a 1 GHz frontside bus per processor, and 1024 MB of RAM. The system runs Max OS X v 10.4.10, kernel version 8.10.0. Algorithms use the OpenSSL cryptographic library (version 0.9.8g) with a modulus N of size 1024 bits and 4 KB file blocks. The experiments were performed using a Maxtor 6Y160M0 SATA 7200 rpm hard disk with 160 GB capacity running the HFS filesystem. Purely for evaluation purposes, we instantiated E_K with AES using 128-bit keys to encrypt the blocks for both MR-PDP and ENC-PDP, and we used SHA1 for h and set $\epsilon = 0$ for the PRF ψ. The random values used for masking in MR-PDP were generated using the key derivation function proposed by Shoup [25].

Experiments measure the processing costs of the different schemes by measuring the computation time in the OpenSSL library. These costs do not include time to perform disk I/O. All data represent the mean of 20 trials. We do not include confidence intervals as the variation between experiments was negligible.

5.1 Pre-Processing Multiple Replicas

We study the scalability of MR-PDP when creating multiple replicas and compare its performance to that of ENC-PDP. Pre-processing is the fundamental performance concern in MR-PDP and the original motivation for this work. The original single-replica PDP scheme [4] provides an extremely efficient challenge phase, but is performance-limited by the time to pre-process one file prior to storage. This limitation is mitigated by the fact that pre-processing is only done once, while challenges are frequent over a long period of storage. However, in the multiple replica setting, the system needs to initially create several replicas, and later to create new replicas on demand; this potentially worsens the limitations of pre-processing.

Our pre-processing experiments show that MR-PDP reduces overall pre-processing costs by amortizing these costs over multiple replicas. This is our principal performance improvement. Fig. 5 gives the per-replica cost of pre-processing a 1 MB file as a function of the number of replicas for MR-PDP, ENC-PDP, and re-replicating with MR-PDP. Re-replicating measures the cost of creating a new replica given that the tags for existing replicas have already been created. It consists of unmasking the file and re-masking it with new randomness.

For a single replica, MR-PDP and ENC-PDP take the same amount of time, because they perform essentially the same tasks (encryption and the generation of tags). ENC-PDP realizes no benefit from generating more replicas and each replica takes less that 9 seconds to pre-process. Re-replicating with MR-PDP measures the cost to unmask and re-mask data only. No new verification tags need to

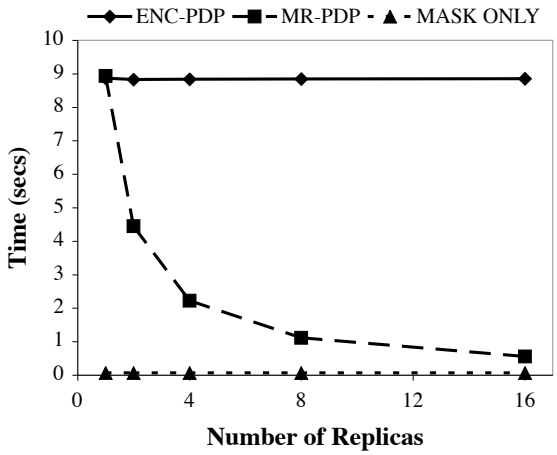

Figure 5. Per-replica computational cost of pre-processing a 1 MB file using ENC-PDP, MR-PDP, and re-replicating with MR-PDP (MASK ONLY).

be computed. Each replica takes only about 0.07 seconds. As we increase the number of replicas, the per-replica cost of MR-PDP decreases. The expensive step of generating the verification tags is amortized over all replicas. In the limit, the performance of MR-PDP approaches that of re-replication. For 16 replicas, MR-PDP takes only 0.56 seconds per replica.

5.2 Challenging Multiple Replicas

We now turn to the performance of MR-PDP in the challenge phase. Fig. 6 shows the client and server processing costs when challenging 460 blocks from a single 100 MB replica. This challenge represents 99% confidence that less than 1% of the data have been damaged [4].

MR-PDP places no additional burden on the storage server during the challenge phase. The server protocol is identical for both MR-PDP and ENC-PDP and, thus, server processing costs are the same. Exponentiating to the sum of the data comprises the entire 0.55 seconds, as the server exponentiates to a number slightly larger than the block size (*i.e.*, 4096 bytes + $\log_2(460)$ bits). The costs of summing and multiplying the information are neglibible. Minimizing the server's effort is essential given that many clients share a server's resources. In fact, limiting I/O and computation costs at the server is one of the principal benefits of the single-replica PDP scheme [4] on which ENC-PDP relies. MR-PDP inherits this benefit.

Challenges incur relatively more cost at the client for MR-PDP than for ENC-PDP. The reason is that the client has the additional cost of generating and exponentiating to the masking information ($g^{r_{u,2}+r_{u,4}+r_{u,7}}$ in the example in Fig.4), which costs an additional 0.111 seconds. This makes

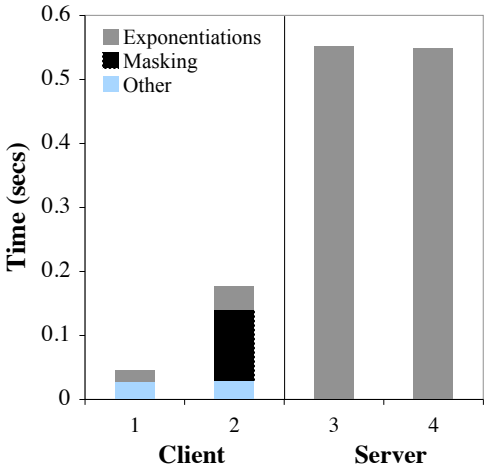

Figure 6. Challenge performance of ENC-PDP and MR-PDP for a single replica.

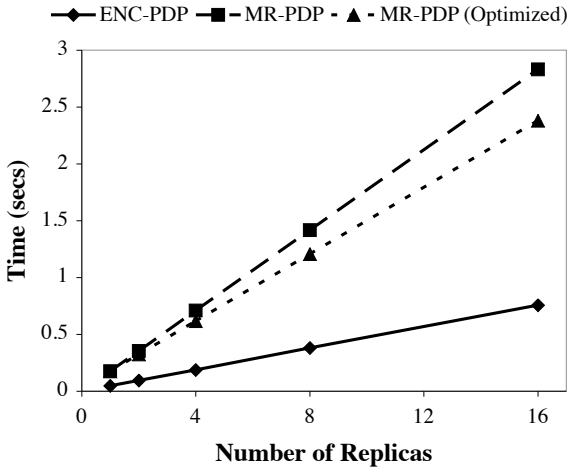

Figure 7. Challenge performance of ENC-PDP and MR-PDP with and without multiple-replica optimizations.

the MR-PDP client computation almost four times as expensive as ENC-PDP: 0.047 seconds versus 0.177 seconds.

The additional computational cost at the client is not a performance or scalability concern when deploying MR-PDP in distributed storage systems. The absolute time spent on computation is still much less than that of the server. The overheads are small and client computation is a scalable resource.

The increased computational demand on the client can be mitigated by conducting concurrent challenges against multiple replicas. In doing so, the client computes a portion of the verification step only once and amortizes this cost over all replicas. Informally, a concurrent challenge breaks the verification computation into two components, which we illustrate using the example in Fig. 4: (1) the data component ($g^{b_2+b_4+b_7}$) is obtained from the aggregated tag ($T^e/(h_2 \cdot h_4 \cdot h_7)$); (2) the masking component ($g^{r_{u,2}+r_{u,4}+r_{u,7}}$). If we use the same indices for the challenged blocks across replicas, then the data component needs to be computed only once for all replicas, whereas the masking component differs for every replica. Fig. 7 shows the relative cost of conducting challenges using ENC-PDP, MR-PDP individually on each replica, and MR-PDP using this optimized concurrent challenge against all replicas. The optimization for checking multiple replicas results in modest, although not insubstantial, savings. At 16 replicas, concurrent checking reduces costs by about 16%. Again, generating the randomness dominates the cost of checking and this component of checking needs to be performed for each replica.

6 Related Work

Replica Maintenance. Previous work on the maintenance of replicas tracks multiple copies of a file throughout a distributed storage system and then creates or destroys replicas in order to achieve management goals. These goals include availability [9], preservation [20], and censorship resistance [28]. None of the previous work on replica maintenance includes guarantees that multiple copies of data are actually maintained.

Maintenance is driven by replication analysis, which characterizes the failure and recovery of data in distributed systems and quantifies the effects on the availability of data. These include birth/death models [11, 29], replica turnover or "churn" [27], and the lifetime of data [22]. The Farsite file system uses a peer-to-peer replica exchange algorithm in order to balance the availability of all replicas in a serverless file system [2, 9]. Lots of Copies Keep Stuff Safe (LOCKSS) [5, 20] employs reputation protocols, called opinion polls, to identify and repair damaged replicas. Opinion polls compare the contents of multiple replicas against each other and, thus, cannot be used to determine that replicas are unique and differentiable.

File Checking. Our work on checking replicas extends the PDP [4] protocol for data checking. We select PDP because of its security and performance parameters. Recently, remote file checking has received much attention.

Juels and Kaliski [17] describe *Proofs of Retrievability* (PoR) for checking that a remote server can produce a file that was previously stored. PoR is based on encrypting a file and inserting random sentinel blocks–that the server cannot differentiate from encrypted blocks—within the file. PoR supports only a finite number of challenges, because each challenge consumes sentinel blocks.

Shah et al. [24] describe a data checking scheme based on data commitment, storing an encrypted file and a one-way function of the encryption key, and checking the com-

mitted version against pre-computed challenges. Because challenges are pre-computed, the scheme also supports only a finite number of challenges and requires metadata linear in the number of challenges.

Deswarte et al. [13] and Filho et al. [14] provide techniques to verify that a remote server stores a file using RSA-based hash functions. The limitation of these algorithms lies in the computational complexity at the server, which must exponentiate the entire file, accessing all of the file's blocks.

Schwarz and Miller [23] present a data checking scheme for distributed erasure-coded data that realizes availability benefits of replication. They use xor-based, parity erasure codes to create n shares of a file that they store at multiple sites. The technique cross compares the contents of the shares using algebraic signatures, which have the property that the signature of the parity block equals the parity of the signatures of the data blocks. The limitations of the Schwarz and Miller scheme lie in the underlying data checking protocol. The file access and computation complexity at the server and the communication complexity are all linear in the total number of file blocks per challenge.

Acknowledgments. We thank Breno de Medeiros and Samuel Wagstaff, Jr. for their insightful comments.

References

[1] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proc. of SOSP '01*, 2001.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI '03*, 2003.

[3] Amazon Simple Storage Service (Amazon S3). `aws.amazon.com/s3`.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS '07*. Full version: Cryptology ePrint Archive. Report 2007/202, 2007.

[5] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. of EuroSys*, 2006.

[6] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. of ACM CCS '93*, 1993.

[7] M. Bellare and P. Rogaway. The exact security of digital signatures - How to sign with RSA and Rabin. In *Proc. of EUROCRYPT*, 1996.

[8] M. Bellare and P. Rogaway. PSS: Provably secure encoding method for digital signatures. IEEE P1363a: Provably secure signatures, 1998.

[9] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasability of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of ACM SIGMETRICS*, 2000.

[10] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI '06*, 2006.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP '01*, 2001.

[12] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. of NSDI '04*, 2004.

[13] Y. Deswarte, J.-J. Quisquater, and A. Saidane. Remote integrity checking. In *Proc. of Conference on Integrity and Internal Control in Information Systems (IICIS'03)*, 2003.

[14] D. L. G. Filho and P. S. L. M. Baretto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, 2006. Report 2006/150.

[15] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Proc. of Financial Cryptography*, 2002.

[16] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI '05*, 2005.

[17] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS*, October 2007.

[18] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of OSDI 2004*.

[19] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *Proc. of USENIX Technical Conference*, 2003.

[20] P. Maniatis, M. Roussopoulos, T. Giuli, D. Rosenthal, M. Baker, and Y. Muliadi. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computing Systems*, 23(1):2–50, 2005.

[21] A. A. Muthitacharoen, R. Morris., T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI '02*, 2002.

[22] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *Proc. of INFOCOM*, 2006.

[23] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proc. of ICDCS*, 2006.

[24] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *Hot Topics in Operating Systems (HotOS)*, 2007.

[25] V. Shoup. A proposal for an ISO standard for public key encryption (v. 2.1). IBM Zurich Research Lab Technical Report, December 2001.

[26] SRB—Storage Resource Broker. `http://www.sdsc.edu/srb/index.php/Main_Page`.

[27] K. Tati and G. Voelker. On object maintenance in peer-to-peer systems. In *International Workshop on Peer-to-Peer Systems*, 2006.

[28] M. Waldman and D. Mazières. Tangler: a censorship-resistant publishing system based on document entanglements. In *Proc. of ACM CCS*, 2001.

[29] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, 2002.