Towards Server-side Repair for Erasure Coding-based Distributed Storage Systems

Bo Chen
Computer Science Department
Stony Brook University
bochen1@cs.stonybrook.edu

Anil Kumar Ammula, Reza Curtmola
Department of Computer Science
New Jersey Institute of Technology
{aa654, crix}@njit.edu

ABSTRACT

Erasure coding is one of the main mechanisms to add redundancy in a distributed storage system, by which a file with k data segments is encoded into a file with n coded segments such that any k coded segments can be used to recover the original k data segments. Each coded segment is stored at a storage server. Under an adversarial setting in which the storage servers can exhibit Byzantine behavior, remote data checking (RDC) can be used to ensure that the stored data remains retrievable over time. The main previous RDC scheme to offer such strong security guarantees, HAIL, has an inefficient repair procedure, which puts a high load on the data owner when repairing even one corrupt data segment.

In this work, we propose RDC-EC, a novel RDC scheme for erasure code-based distributed storage systems that can function under an adversarial setting. With RDC-EC we offer a solution to an open problem posed in previous work and build the first such system that has an efficient repair phase. The main insight is that RDC-EC is able to reduce the load on the data owner during the repair phase (i.e., lower bandwidth and computation) by shifting most of the burden from the data owner to the storage servers during repair. RDC-EC is able to maintain the advantages of systematic erasure coding: optimal storage for a certain reliability level and sub-file access. We build a prototype for RDC-EC and show experimentally that RDC-EC can handle efficiently large amounts of data.

1. Introduction

Remote data checking (RDC) [4, 24, 27, 7, 3] allows data owners to check the integrity of data stored at an untrusted server, thus enabling data owners to audit whether the server fulfills its contractual obligations. Long-term storage usually imposes certain reliability guarantees, such that the data remains retrievable over time. To achieve this guarantee, a distributed storage system usually stores data redundantly at multiple servers which are geographically spread throughout the world. In a benign setting where the storage servers are trusted, this basic approach would be sufficient to handle server failure due to natural faults. However, in an adversarial setting where the storage servers are untrusted and may behave maliciously, the basic approach may not be able to provide the desired reliability guarantee over time. In an adversarial setting, RDC can be used to ensure that valuable data is retrievable over time even if the storage servers are malicious.

When a distributed storage system is used in tandem with remote data checking, we distinguish several phases throughout the storage system's lifetime: Setup, Challenge, and Repair. During Setup, the data owner stores data redundantly at multiple storage servers. During Challenge, the data owner checks periodically each storage server to ensure the data stored at each server remains intact. If the data at one of the servers is found corrupted, the Repair phase is activated and the data owner repairs the data at the corrupted server using data from the healthy servers, such that the desired redundancy level in the system is restored. The Challenge and Repair phases will alternate over the lifetime of the storage system.

In a benign setting, the methods for storing data redundantly at multiple servers fall under three categories: *replication*, *erasure coding*, and *network coding*. We give an overview of these methods in Sec. 2, but we make here a few important observations. From a storage perspective, erasure coding is optimal since it can achieve the same reliability as replication at only a fraction of the storage cost. However, from the perspective of communication overhead in the repair phase, network coding is optimal as it incurs only a fraction of the communication imposed by erasure coding. Still, network coding has a major drawback which limits its applicability: small portions cannot be read without reconstructing the entire file. The fact that network coding does not support sub-file access to data makes it unsuitable for applications in which data read is a frequent operation.

Current RDC schemes designed for replication (MR-PDP [16]), erasure coding (HAIL [7]), and network coding (RDC-NC [15]) incur storage and communication costs as described in Table 1. We can see that these RDC schemes (approximately) preserve in an adversarial setting the storage and communication parameters that characterize a benign setting. Yet, we notice that if we were to use HAIL (*i.e.*, the RDC scheme which minimizes the storage cost and allows sub-file access), then we would have to pay the highest communication cost during the Repair phase: Repairing data at one corrupted server requires the data owner to retrieve all the data at all the storage servers, reconstruct the entire data and then recompute the corrupted segment. This process may put a high burden on the data owner. In fact, the design of an RDC scheme for erasure coding-based distributed storage systems with low-bandwidth repair was posed as an open problem [7] and remained unsolved.

In this work, we provide a solution to this open problem by designing RDC-EC, a distributed storage system which functions under an adversarial setting and achieves both the storage benefits of erasure coding and the repair bandwidth benefits of network coding. RDC-EC has the following properties:

– It minimizes the storage cost (in order to achieve a certain reliability level).
– It allows efficient sub-file access.
– It incurs low repair bandwidth between the data owner and the storage servers.
– It functions properly under an adversarial setting.

Table 1 compares the performance of our scheme (RDC-EC) with previous RDC schemes.

1.1 Solution Overview

	MR-PDP [16]	HAIL [7]	RDC-NC [15]	RDC-EC (proposed approach)
Coding method	replication	erasure coding	network coding	erasure coding
Total server storage	$O(n\lvert\mathtt{F}\rvert)$	$O(\frac{n\lvert\mathtt{F}\rvert}{k})$	$O(\frac{2n\lvert\mathtt{F}\rvert}{k+1})$	$O(\frac{n\lvert\mathtt{F}\rvert}{k})$
Communication (repair phase)	$O(\lvert\mathtt{F}\rvert)$	$O(\frac{n\lvert\mathtt{F}\rvert}{k})$	$O(\frac{2\lvert\mathtt{F}\rvert}{k+1})$	$O(\frac{\lvert\mathtt{F}\rvert}{k})$
Total server computation (repair phase)	$O(1)$	$O(1)$	$O(\ell)$	$O(k)$
Support for sub-file access	yes	yes	no	yes

Table 1: A comparison of RDC schemes for distributed storage systems. A file $\mathtt{F}$ of $\lvert\mathtt{F}\rvert$ bits has originally k segments and is encoded into n segments. For the repair phase, the costs are for the case when one storage server fails. For RDC-NC, the client retrieves data from ℓ servers during Repair.

Our starting point is the HAIL scheme [7], which views the original data (e.g., a file) as a collection of k segments and encodes it into a collection of n segments. The n coded segments are stored at n storage servers (one segment per server). HAIL is designed to withstand a Byzantine and mobile adversary which can corrupt at most b servers in any time interval (i.e., an *epoch*). Because, in time, all the n storage servers could be corrupted, the servers are periodically challenged to provide a proof that they continue to store data. If a server is found faulty, then a repair procedure is triggered in order to bring back the system to a state in which all data is recoverable. Whereas HAIL is designed to withstand attacks while minimizing overall storage costs and communication costs during the Challenge phase, its Repair phase is inefficient: to repair even one corrupt segment, the data owner has to retrieve all the n segments from the n servers, reconstruct the entire file, and then recompute the corrupted segment.

We inherit from HAIL the optimal storage cost and low communication cost during the Challenge phase. However, we redesign parts of HAIL to achieve an efficient Repair phase, in which the repair bandwidth is (asymptotically) equal to the optimal repair bandwidth. The design of our new scheme is motivated by two insights:

Insight 1. Server-side repair: We leverage server-side repair [13], a recently proposed concept which can minimize the load on the data owner during the Repair phase by allowing the storage servers to collaborate in order to generate a new segment whenever an existing segment has been corrupted. By incorporating server-side repair into RDC-EC, we obtain the following advantages:

– the repair bandwidth between the data owner and the storage servers is reduced considerably (only two segments are transmitted instead of n segments like in HAIL). The majority of the data transmission during Repair now happens between the storage servers. This is beneficial since the data owner's connection may have limited bandwidth, whereas the storage servers are usually connected by a high bandwidth network.
– the computational burden during the Repair phase is shifted to the servers, allowing data owners to remain lightweight.

Insight 2. The elements of the encoding matrix are masked: To enable server-side repair, previous work reveals certain secrets to allow the servers to collaborate and repair the corrupted data. For example, when the distributed storage system relies on replication [13], the data owner reveals to the servers the secret key needed to differentiate the various replicas. A straightforward extension to the setting of erasure coding, would mean that the data owner must reveal the encoding matrix (*i.e.*, the matrix used to erasure code the original data). In all previous RDC schemes for erasure coding-based distributed systems (HAIL [7] and [31]), this encoding matrix needs to remain secret, otherwise the data can be corrupted unbeknownst to the data owner (such an attack is described in Sec. 4.3).

To overcome this potential attack, the data owner does not reveal the encoding matrix to the storage servers. Instead, to repair a corrupt segment, the data owner engages in a two-round protocol with the storage system as follows. The data owner masks certain elements of the encoding matrix and provides them to the storage servers. These masked elements do not reveal anything about the original elements in the encoding matrix, but are used by the servers to collectively perform blind computations over the segments they store and to obtain two masked segments. The data owner receives the two masked segments and has enough information to unmask and combine them into one segment. This segment is sent to a new storage server to replace the corrupt segment.

This approach has the additional advantage of reducing the computational load on data owners. Instead of participating in an expensive decoding of n segments to reconstruct the entire data (like in HAIL), the data owner only processes two segments and lets the servers handle the bulk of the decoding and reconstruction.

What are the trade-offs? RDC-EC is able to reduce the load on the data owner during Repair (*i.e.*, lower bandwidth and computation) by shifting most of the burden from the data owner to the storage servers. Basically, RDC-EC incurs the following costs in order to achieve a more efficient Repair phase:
– increased load and more complex functionality at the storage servers (the servers are required to perform computations during Repair, as opposed to simply "serving" their data segments in HAIL; the servers are also required to perform additional interactions with the data owner and with the other servers during Repair).
– two rounds of interaction between the data owner and the storage system (as opposed to one round in HAIL).

We believe this trade-off between the resources of the data owner and those of the storage servers aligns well with the cloud computing model which assumes a resource-rich cloud data center.

1.2 Contributions

In this work, we propose RDC-EC, a remote data checking scheme for erasure code-based distributed storage systems that can function under an adversarial setting. With RDC-EC we offer a solution to an open problem posed in previous work and build the first such system that has an efficient Repair phase. The main insight is that RDC-EC is able to reduce the load on the data owner during Repair (i.e., lower bandwidth and computation) by shifting most of the burden from the data owner to the storage servers. Specifically, we make the following contributions:

- We provide an overview of the main methods to add redundancy in a distributed storage system. The previous main result that uses remote data checking (RDC) to ensure data remains retrievable over time in an erasure coding-based distributed storage system is HAIL [7]. Whereas HAIL is able to provide security guarantees against a strong adversary, it has an inefficient Repair phase.

- We propose RDC-EC, a remote data checking scheme for erasure code-based distributed storage systems that can function under an adversarial setting. With RDC-EC we offer a solution to an open problem posed in HAIL and build the

first such system that has an efficient Repair phase. Similar with HAIL, RDC-EC inherits the advantages of systematic erasure coding: the storage overhead is optimal in order to achieve a certain reliability level, and sub-file access is possible (*i.e.*, small portions of the coded data can be read without having to decode the entire data). Like HAIL, RDC-EC handles a strong mobile and Byzantine adversary which is allowed to corrupt b out of the n storage servers in any time interval (*i.e.*, an epoch). However, unlike HAIL, RDC-EC is able to repair one corrupt segment by placing a minimal load on the data owner who only needs to download and process the equivalent of two segments (whereas HAIL requires n segments to be downloaded and processed).
RDC-EC achieves efficient repair by incorporating *server-side repair*, which can minimize the load on the data owner during the Repair phase by allowing the storage servers to collaborate in order to generate a new segment whenever an existing segment has been corrupted. The main challenge is to leverage the resources of the storage servers while not revealing information that would allow the adversarial servers to attack the system and reduce its reliability over time. The data owner achieves this by requiring the servers to perform computation over a "masked" version of the data, which is then "un-masked" and used to recover the corrupted segment.

- We build a prototype for RDC-EC using the Jerasure [26] and OpenSSL [2] libraries. To achieve an appropriate security level, we extended Jerasure's coding functions to support 128-bit symbols. Experimental evaluations show that RDC-EC is efficient for encoding and repairing large files (less than 20 sec for a 1GB file). Due to space limitations, implementation and experimental details are presented in [10].

2. Redundancy in Distributed Storage Systems

Data can be redundantly stored at multiple servers through *replication*, *erasure coding*, and *network coding*.

Replication. In replication, the data owner simply stores multiple copies of a file at multiple storage servers, such that the file is recoverable if at least one file copy remains intact. Although replication has the advantage of simplicity, it has a high storage cost. Today's storage systems usually handle "big data", which can be TBs or even PBs. Replicating multiple entire copies of the "big data" may become prohibitively expensive.

Erasure coding-based distributed storage systems. In erasure coding, given a file of k segments, the client encodes the file into n coded segments, such that any k out of n coded segments can be used to restore the original file. The client stores the n coded segments at n servers, one segment at each server. We provide details about the encoding/decoding process in [10].

Erasure coding was shown to be optimal in terms of redundancy-reliability tradeoff [34] and has been used extensively to ensure reliability for storage systems [9, 23, 26]. In addition, an erasure code is systematic, with its input embedded as part of its encoded output, *e.g.*, in an (n, k) erasure code, the first k coded segments are the k original file segments. This has the advantage that any portion of the file can be read efficiently (we call this property "*sub-file access*"). Due to the aforementioned advantages, erasure coding was used broadly in storage systems which require frequent reads and can be categorized as read-frequently workloads, *e.g.*, Microsoft Azure [1], HYDRAstor [19], etc.

Bowers et al. [7] introduced HAIL, a distributed cloud storage system which offers cloud users high reliability guarantees under a strong adversarial setting. Similar to RAID [25], which builds low-cost reliable storage from inexpensive drives, HAIL builds reliable cloud storage by combining cheap cloud storage providers. However, RAID has been designed to tolerate benign failures (*e.g.*, hard drive crashes), whereas HAIL is able to deal with a strong (*i.e.*, mobile and Byzantine) adversarial model, in which the adversary is allowed to perform progressive corruption of the storage providers over time. We provide an overview of HAIL in [10].

Unfortunately, erasure coding has an inefficient repair procedure. Repairing one corrupted segment requires to download k coded segments and reconstruct the whole file. This is exacerbated in an adversarial setting: to repair a corrupted segment, the client needs to download all the n coded segments (*e.g.*, HAIL [7])).

Network coding-based distributed storage systems. Network coding can be used to encode and distribute a file to n servers, such that any k out of n servers have enough data to recover the file [17, 18]. Although network coding can achieve optimal repair bandwidth, it is not systematic. This makes it inefficient for read operations, because reading even a small portion of the file requires to reconstruct the whole file. Thus, applications of network coding to storage systems are limited to read-rarely workloads [15].

3. System and Adversarial Model

The client (*i.e.*, data owner) divides the original file into k segments, and encodes them into n coded segments, such that: 1) the first k coded segments are the original file segments, and the remaining $n - k$ coded segments are the parity segments; 2) any k out of n segments can be used to restore the original file. The client stores the n coded segments at n storage servers (one segment per server) as follows: The k original segments are stored at k primary servers $(S_1, S_2, \ldots, S_k)$, whereas the $n - k$ parity segments are stored at $n - k$ secondary servers $(S_{k+1}, \ldots, S_n)$.

We consider a *mobile* and *Byzantine* adversary, similar to the one used in HAIL [7]. "Byzantine" means the adversary can behave arbitrarily. "Mobile" means that the adversary can corrupt any (and potentially all) of the servers over the lifetime of the storage system. However, it can only corrupt at most $(n - k - 1)/2$ out of the n servers within any given time interval. We refer to such a time interval as an *epoch*.

From an adversarial point of view, a storage server is seen as having two components, the *code* and the *storage*. The code refers to the software that runs on the server and defines the server's behavior in the interaction with the client, whereas the storage refers to the data stored by the server.

At the beginning of each epoch, the adversary picks a set of at most $(n - k - 1)/2$ servers, and corrupts both the code and the storage components on them. At the end of each epoch, the code component in each storage server will be restored to a correct state (*e.g.*, the data owner can simply remove the malware, and re-install a clean code component). However, the storage component may remain corrupted across epochs. Thus, in the absence of explicit defense mechanisms, the storage at more than $n - k$ servers may become corrupted over time and the original data may become unrecoverable. The client's goal is to detect and repair storage corruption before it renders the data unavailable. To this end, the client checks data possession with the servers in every epoch and if it detects any corrupted data in the faulty servers, it uses the redundancy at the remaining healthy servers to repair the corruption.

Each epoch consists of two phases:

- A *challenge phase* that contains two sub-phases:
  (a) corruption sub-phase: The adversary corrupts up to b_1 servers.
  (b) challenge sub-phase: The client checks data possession with all the servers. As a result, it may detect faulty servers with corrupted data (*i.e.*, faulty servers).
- A *repair phase* that contains two sub-phases and is triggered only if corruption is detected in the challenge phase:
  (a) corruption sub-phase: The adversary corrupts up to b_2 servers.
  (b) repair sub-phase: The client repairs the data at any faulty servers detected in the challenge phase.

The total number of servers that can be corrupted by the adversary during an epoch is at most $(n - k - 1)/2$ (*i.e.*, $b_1 + b_2 \leq (n - k - 1)/2$).

The structure of an epoch is similar with the one in HAIL [7], with one important modification: We explicitly allow the adversary to corrupt data after the challenge phase. This models attackers that act honestly during the challenge phase, but are malicious in the repair phase. Such behavior must be considered in any scheme that seeks to achieve a repair phase that is more efficient than simply retrieving all the n segments.

4. Remote Data Checking for Erasure Coding-based Distributed Storage Systems

In this section, we first introduce the key ideas in our design, and then present our main result, the RDC-EC scheme.

4.1 Key Ideas

A novel repair tag construction. We introduce a novel repair tag construction in which, to compute a repair tag for an erasure coded segment, we use a seed to generate a set of pseudorandom coefficients, and then use this set of coefficients to aggregate the symbols in the segment. A repair tag constructed in this way is *publicly computable*, but *privately verifiable*. Publicly computable means any party who knows the seed can generate the corresponding repair tag for the stored segment. Private verifiable means the correctness of the repair tags can be verified only by a party who knows some secret. During Repair, each storage server computes a repair tag for its stored segment based on a fresh seed sent by the client (publicly computable). The client can then verify the correctness of the repair tags from all the servers based on the seed and on a secret (privately verifiable), because all the repair tags together form a valid integrity-protected dispersal code that was constructed based on the client's secret (for details about the integrity-protected dispersal code, refer to HAIL [7]).

Enabling server-side repair for erasure coding to minimize the client's workload. In previous erasure coding-based distributed storage systems such as HAIL, the original file of k segments is encoded into a code word of n segments which are outsourced to n storage servers. To repair a corrupted segment, the client first retrieves all the outsourced segments, decodes them to recover the original k file segments and then restores the corrupted segment. Our key idea is to leverage *server-side repair*, in which we allow the storage servers to collaborate in order to generate a new segment whenever an existing segment has been corrupted. As a result, the communication and computation load on the client during Repair is reduced considerably.

Enabling server-side repair to minimize the client's load in the setting of erasure coding-based distributed systems imposes several design decisions. During Repair, we require the client to verify the repair tags because this verification relies on secret keys and cannot be offloaded to the untrusted servers. The client needs to retrieve n small tags and perform a small amount of computation on them, which impose only a small burden on the client. In addition, to protect secret keys used to embed random values into the parity symbols (for the integrity-protected dispersal code), we require the client (rather than the servers) to perform the final step of restoring the corrupted segment. In this way, the client can remain lightweight because it only needs to perform a small amount of computation over a limited number of segments (*i.e.*, two in RDC-EC) in order to restore a corrupted segment. Most of the repair work (*e.g.*, aggregating a large number of segments to decode the original file) is offloaded to the servers.

Allowing untrusted servers to aggregate segments under an adversarial setting. In a benign setting, the information dispersal matrix used to erasure code the original file is needed to repair a corrupted segment. However, under an adversarial setting, this matrix needs to remain secret during the server-side repair process. Thus, the client has to leverage the computational power of the servers without revealing this matrix. For this, the client derives a set of intermediate coefficients from the dispersal matrix, masks them based on an algebraic function and sends them to the untrusted servers. Due to the algebraic properties of our masking function and of the erasure coding, the servers are able to perform useful computation over their stored segments based on the masked coefficients. This approach ensures the secrecy of the information dispersal matrix while allowing server-side repair.

4.2 The RDC-EC scheme

In the following, we present RDC-EC, the first erasure coding-based remote data checking scheme that allows server-side repair. This is the main result of the paper.

Let κ be a security parameter. The original file $\mathtt{F}$ is divided into k segments: $\mathtt{F} = \{\mathtt{b}_1, \ldots, \mathtt{b}_k\}$. Each segment $\mathtt{b}_i$ can be viewed as a column vector: $\mathtt{b}_i = (\mathtt{b}_{i1}, \mathtt{b}_{i2}, \ldots, \mathtt{b}_{i\ell})$, where $\mathtt{b}_{ij}$ $(1 \leq j \leq \ell)$ is a symbol in $GF(2^w)$ and ℓ is the number of symbols in a segment. Throughout the paper, all the arithmetic operations are performed in $GF(2^w)$. We make use of a PRF g with the following parameters: $g : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow GF(2^w)$. We use $file\_handle$ to uniquely identify the file to be encoded.

RDC-EC overview. Our RDC-EC scheme consists of three phases: Setup, Challenge, and Repair. During the Setup phase, the client preprocesses the original file $\mathtt{F}$. The client first generates an $n \times k$ information dispersal matrix M and then uses M to encode the k segments of $\mathtt{F}$, generating n coded segments such that: (a) the first k coded segments are the original file segments, (b) the remaining $n - k$ coded segments are the parity segments, and (c) any k out of n coded segments can be used to recover $\mathtt{F}$. This part of the encoding process is described in more details in Sec. 2 and in [10]. The client further adds to each parity symbol a secret value, such that each parity symbol is converted into a message authentication code (MAC) of the corresponding file symbols [7]. The client then stores the n segments at n servers (one segment per server).

In the Challenge phase, the client challenges each of the n servers, requiring them to prove data possession of the stored segments. This integrity check can be achieved efficiently based on spot checking, in which the client only checks a random subset of symbols from each outsourced segment. Prior work shows that spot checking provides a probabilistic guarantee for corruption detection, but the detection probability can be made arbitrarily high by increasing the number of symbols being challenged [4]. In RDC-EC, the client challenges the same random subset of symbols from each of the n stored segments. Each server aggregates the challenged symbols based on the same set of random coefficients and sends back an aggregated response. After having received all the n aggregated responses, the client can check and localize the corrupted responses, because the n responses constitute a valid integrity-protected dispersal code. The Challenge phases in RDC-EC and HAIL [7] are similar.

The Repair phase is activated when the client detected at least one corrupted segment during Challenge. We do not differentiate between corruptions caused by malicious server behavior and natural faults because servers which allow natural faults to be visible to clients should be avoided just as well. To repair a corrupted segment, the client randomly picks a seed and sends it to the n servers. Each server computes and sends back a *repair tag*, which is generated based on the stored segment and the random seed. After receiving the n repair tags, the client verifies all the n repair tags, and is able to detect and localize any corrupted repair tags because the n repair tags constitute a valid integrity-protected dispersal code (see Sec. 4.1). The client then picks k repair servers, each of which will provide data for repairing the corrupted segment. Note that when picking the repair servers, the client will exclude the servers which were found corrupted either in the Challenge phase or in the aforementioned repair tag verification. From the remaining $n-k$ servers, the client randomly picks an *Aggregation Server* (AS), who will be responsible for aggregating the data provided by the repair servers. The following steps occur next:

1. Based on the information dispersal matrix M, the client generates a set of intermediate coefficients. These coefficients are used to linearly combine the segments stored in the repair servers in order to recover the corrupted segment. To avoid leaking information about M, the intermediate coefficients are masked using secret random values before being sent to the repair servers.

2. Each repair server computes two partial segments based on its stored segment and the masked coefficients sent by the client. A *partial segment* is computed by multiplying the stored segment with a masked coefficient provided by the client.

3. The client restores AS's code component[1]. Since the AS's code component has been restored, we assume the AS acts honestly until the end of the Repair phase. The client sends to the AS both the masked intermediate coefficients and the k repair tags corresponding to the k repair servers.

4. Each repair server sends its two partial segments to the AS.

5. AS aggregates the partial segments, generating two intermediate segments. It verifies whether the intermediate segments are correctly computed based on the repair tags sent by the client. If the verification fails, it can further localize the faulty servers based on these repair tags. Ultimately, two correctly computed intermediate segments are sent back to the client.

6. The client uses the intermediate segments and some secret key material to recover the corrupted segment.

The client has to restore the code component on one of the servers (the AS) for a short period of time during the Repair phase. Although this requirement is not present in HAIL's Repair phase, we argue it is reasonable because code restoration is already required for all the servers at the end of each epoch in both HAIL and RDC-EC (see the adversarial model in Sec. 3).

[1]This can be achieved by removing the malware, and re-installing a clean code component. See the adversarial model in Sec. 3.

```
We construct RDC-EC in three phases, Setup, Challenge, and Repair. All the arithmetic operations are performed over the finite field $GF(2^w)$. We use $file\_handle$ to identify the file to be encoded.
Setup: The client $C$ runs sk $\leftarrow$ KeyGen($1^\kappa$). $C$ divides the file F into $k$ segments, $b_1, \ldots, b_k$, then executes:
1. Generate the information dispersal matrix $M$ by running
   $M \leftarrow$ GenInformationDispersalMatrix($sk, k, n$)
2. For $k+1 \le i \le n$:
   Generate the parity segment $b_i$: $(b_{i1}, \ldots, b_{i\ell}) \leftarrow$ ComputeParityHAIL(sk, $b_1, \ldots, b_k, M, i$)
3. For $1 \le i \le n$: Send segment $b_i$ to server $S_i$ for storage
4. Delete the file F and store only the secret key sk
Challenge: Similar with the challenge phase in HAIL: The client cross-checks the $n$ storage servers as described in [7].
```

Figure 1: The Setup and Challenge Phases of RDC-EC

```
KeyGen($1^\kappa$):
1. Choose $2n-k+2$ keys at random from $\{0,1\}^\kappa$:
   $K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b$
2. Return $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
GenInformationDispersalMatrix(sk, $k$, $n$):
1. Parse sk as $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
2. For $1 \le i \le n$
     • For $1 \le j \le k$: $M_{ij} = K_i^j$ (that is, $K_i$ raised to power $j$)
3. Transform $M$ into a matrix in which the first $k$ rows form an identity matrix
4. Return $M$
ComputeParityHAIL(sk, $b_1, \ldots, b_k, M, i$):
1. Parse sk as $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
2. For $1 \le j \le \ell$:
   $b_{ij} = \sum_{\alpha=1}^{k} M_{i\alpha} b_{\alpha j} + g_{K'_i}(file\_handle||i||j)$
3. Return $(b_{i1}, \ldots, b_{i\ell})$
```

Figure 3: Components of the RDC-EC scheme (1)

The RDC-EC scheme. The details of the RDC-EC scheme are presented in Figures 1 (Setup and Challenge) and 2 (Repair), whereas Figures 3 and 4 contain components used in these phases.

The Setup phase. The client first generates keys $K_1, K_2, \ldots, K_n$, $K'_{k+1}, K'_{k+2}, \ldots, K'_n$ at random[2] and then an $n \times k$ information dispersal matrix M by running GenInformationDispersalMatrix. The client further computes n coded segments, in which the first k coded segments are the original file segments, and the remaining $n-k$ coded segments are computed by running ComputeParityHAIL. All the n coded segments are sent for storage at n storage servers, one segment per server. The client may now delete the original file F and only keep the key material.

In GenInformationDispersalMatrix, the client generates the information dispersal matrix M. It first computes an $n \times k$ Vandermonde matrix (as described in [10]), in which each element is computed as K_i^j, where i is the row index ($1 \le i \le n$) and j is the column index ($1 \le j \le k$). The client then transforms this matrix into a matrix whose first k rows form the identity matrix.

In ComputeParityHAIL, the client computes a parity segment with index i, where $k+1 \le i \le n$, and embeds a MAC into each symbol of this segment. The client uses the set of elements in

[2]Note that we can save storage by using a PRF and a master key to generate all these keys on the fly when needed.

Repair: Assume during Challenge the client C has detected a corrupted segment $\mathtt{b}_y$ and has identified the corresponding faulty server S_y.

1. C generates a key K at random from $\{0,1\}^\kappa$. C sends K to each of the n servers.
2. Each of the n servers computes and sends back a repair tag:
   For $1 \leq i \leq n$: S_i runs $t_i \leftarrow \mathsf{ComputeRepairTag}(K, \mathtt{b}_i)$, and sends t_i back to C
3. C verifies all the repair tags received from the n servers by running
   $(G, flag) \leftarrow \mathsf{VerifyAllRepairTag}(\mathrm{sk}, t_1, t_2, \ldots, t_n, K, M, n, k)$. If $flag = 0$, exit (the file cannot be repaired, too many servers are faulty).
4. C chooses k different repair servers $i_1, \ldots, i_k$, each of which is randomly picked from the healthy servers (*i.e.*, excluding server S_y and the servers in G which were found to have sent invalid repair tags). Note that $i_1, \ldots, i_k$ are in ascending order. C then computes the set of *intermediate coefficients* $z_{i_1}, \ldots, z_{i_k}$: $(z_{i_1}, \ldots, z_{i_k}) \leftarrow \mathsf{GenRepairServerCoefficient}(\mathrm{sk}, y, k, n, i_1, \ldots, i_k)$.
5. C generates $k+2$ random numbers $a, r, x_1, \ldots, x_k$ by running $(a, r, x_1, \ldots, x_k) \leftarrow \mathsf{GenRandom}(\mathrm{sk}, y, i_1, i_2, \ldots, i_k)$, and masks $z_{i_1}, \ldots, z_{i_k}$: For $1 \leq j \leq k$: $Z_{i_j} = az_{i_j} + rx_j$
6. C sends to the repair servers the *masked coefficients* $(Z_{i_1}, Z_{i_2}, \ldots, Z_{i_k}, x_1, x_2, \ldots, x_k)$
7. Each of the k repair servers computes two partial segments:
   For $1 \leq j \leq k$:
   (a) S_{i_j} computes a first partial segment $\mathtt{b}'_{i_j} = (\mathtt{b}'_{i_j1}, \ldots, \mathtt{b}'_{i_j\ell})$: For $1 \leq \alpha \leq \ell$: $\mathtt{b}'_{i_j\alpha} = Z_{i_j}\mathtt{b}_{i_j\alpha}$
   (b) S_{i_j} computes a second partial segment $\mathtt{b}''_{i_j} = (\mathtt{b}''_{i_j1}, \ldots, \mathtt{b}''_{i_j\ell})$: For $1 \leq \alpha \leq \ell$: $\mathtt{b}''_{i_j\alpha} = x_j\mathtt{b}_{i_j\alpha}$
8. C randomly picks an Aggregation Server (AS) from the remaining $n-k$ servers, and restores the code component at AS (*i.e.*, the AS will behave honestly up to the end of this epoch). C sends $(Z_{i_1}, Z_{i_2}, \ldots, Z_{i_k}, x_1, x_2, \ldots, x_k, t_{i_1}, t_{i_2}, \ldots, t_{i_k}, K)$ to the AS
9. Each of the k repair servers sends the computed partial segments to AS: For $1 \leq j \leq k$: S_{i_j} sends $\mathtt{b}'_{i_j}$ and $\mathtt{b}''_{i_j}$ to AS
10. AS aggregates the partial segments received from the k repair servers and generates two intermediate segments $\mathtt{b}'_y$ and $\mathtt{b}''_y$:
    (a) AS computes the first intermediate segment $\mathtt{b}'_y = (\mathtt{b}'_{y1}, \ldots, \mathtt{b}'_{y\ell})$: For $1 \leq \alpha \leq \ell$: $\mathtt{b}'_{y\alpha} = \sum_{j=1}^{k} \mathtt{b}'_{i_j\alpha}$
    (b) AS computes the second intermediate segment $\mathtt{b}''_y = (\mathtt{b}''_{y1}, \ldots, \mathtt{b}''_{y\ell})$: For $1 \leq \alpha \leq \ell$: $\mathtt{b}''_{y\alpha} = \sum_{j=1}^{k} \mathtt{b}''_{i_j\alpha}$
11. AS computes the repair tags for the two intermediate segments by running $t' \leftarrow \mathsf{ComputeRepairTag}(K, \mathtt{b}'_y)$ and $t'' \leftarrow \mathsf{ComputeRepairTag}(K, \mathtt{b}''_y)$, and checks the correctness of t' and t'':
    (a) If $t' \neq \sum_{j=1}^{k} Z_{i_j}t_{i_j}$, AS localizes the corrupted partial segments among $\mathtt{b}'_{i_1}$, $\mathtt{b}'_{i_2}$, $\ldots$, $\mathtt{b}'_{i_k}$ by running $G \leftarrow \mathsf{VerifyPartialRepairTag}(\mathtt{b}'_{i_1}, \mathtt{b}'_{i_2}, \ldots, \mathtt{b}'_{i_k}, K, t_{i_1}, t_{i_2}, \ldots, t_{i_k}, Z_{i_1}, Z_{i_2}, \ldots, Z_{i_k})$. AS informs C to pick new repair servers to replace the faulty repair servers in G
    (b) Otherwise, if $t'' \neq \sum_{j=1}^{k} x_jt_j$, AS localizes the corrupted partial segments among $\mathtt{b}''_{i_1}$, $\mathtt{b}''_{i_2}$, $\ldots$, $\mathtt{b}''_{i_k}$ by running $G \leftarrow \mathsf{VerifyPartialRepairTag}(\mathtt{b}''_{i_1}, \mathtt{b}''_{i_2}, \ldots, \mathtt{b}''_{i_k}, K, t_{i_1}, t_{i_2}, \ldots, t_{i_k}, x_1, x_2, \ldots, x_k)$. AS informs C to pick new repair servers to replace the faulty repair servers in G
    (c) Otherwise, if $t' = \sum_{j=1}^{k} Z_{i_j}t_{i_j}$ and $t'' = \sum_{j=1}^{k} x_jt_j$, AS sends $\mathtt{b}'_y$ and $\mathtt{b}''_y$ back to C, and C uses them to restores b_y by running $(\mathtt{b}_{y1}, \ldots, \mathtt{b}_{y\ell}) \leftarrow \mathsf{RepairOneSegment}(\mathrm{sk}, \mathtt{b}'_y, \mathtt{b}''_y, i_1, \ldots, i_k, z_{i1}, \ldots, z_{ik}, y, a, r)$. C stores b_y at a new server S'.

Figure 2: The Repair **Phase of** RDC-EC

the i-th row of M to linearly combine the k original file segments, generating the corresponding parity segment. Each symbol in this parity segment is further converted into a MAC (for the k file symbols used to generate this parity symbol) by adding to it a secret random value, which is generated by applying PRF g keyed with K'_i over the concatenation of the unique file handle, the segment index i, and the location j of this symbol in the parity segment.

The Repair **phase.** To repair one corrupted segment, the client first generates a key K at random and sends it to the n servers. Each server S_i computes a repair tag by running ComputeRepairTag, and sends back the repair tag t_i, where $1 \leq i \leq n$. The client verifies the n repair tags $t_1, t_2, \ldots, t_n$ by running VerifyAllRepairTag. If VerifyAllRepairTag returns successfully, the client picks k repair servers, excluding the servers being found corrupted in either the Challenge phase or the aforementioned repair tag verification. It then runs GenRepairServerCoefficient to generate a set of indices $i_1, \ldots, i_k$ and a set of *intermediate coefficients* $z_{i_1}, \ldots, z_{i_k}$. The indices $i_1, \ldots, i_k$ will be used to identify the k repair servers which will provide data for repairing the corrupted segment. The intermediate coefficients $z_{i_1}, \ldots, z_{i_k}$ will be used to linearly aggregate the segments stored in the repair servers to restore the corrupted segment. To prevent the intermediate coefficients from being leaked to the untrusted servers, the client masks each of the intermediate coefficients in the following way: $Z_{i_j} = az_{i_j} + rx_j$, in which a, r, x_j (where $1 \leq j \leq k$) are random values generated by running GenRandom. It then sends the *masked coefficients* $Z_{i_1}, \ldots, Z_{i_k}$ and $x_1, \ldots, x_k$ to the k repair servers. Each repair server S_{i_j} computes two partial segments based on the stored segment $\mathtt{b}_{i_j}$ and the corresponding masked coefficients Z_{i_j} and x_j, where $1 \leq j \leq k$.

Meanwhile, the client picks an Aggregation Server (AS) randomly from the remaining $n-k$ servers. It restores the code component at the AS and discloses to the AS the values $Z_{i_1}, \ldots, Z_{i_k}$, $x_1, \ldots, x_k$, $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ and K. Each repair server then sends the computed partial segments to the AS and the AS aggregates the partial segments, generating two intermediate segments $\mathtt{b}'_y$ and $\mathtt{b}''_y$. AS further computes the repair tags t' and t'' for the intermediate segments by running ComputeRepairTag, and verifies the correctness of t' and t'' using $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$. If the verification succeeds, the AS will send back $\mathtt{b}'_y$ and $\mathtt{b}''_y$ to the client. Otherwise, it runs VerifyPartialRepairTag to localize the faulty repair servers, and informs the client to pick new repair servers to replace the faulty repair servers. After receiving $\mathtt{b}'_y$ and $\mathtt{b}''_y$, the client calls RepairOneSegment to restore the corrupted segment.

In ComputeRepairTag, to generate the repair tag for segment $\mathtt{b}_i$ based on key K, we first generate ℓ random numbers and then use these numbers to linearly aggregate all the symbols in $\mathtt{b}_i$.

```
ComputeRepairTag($K, \mathtt{b}_i$):
1. Return $t = \sum_{j=1}^{\ell}(g_K(file\_handle||j))\mathtt{b}_{ij}$
VerifyAllRepairTag(sk, $t_1, t_2, \ldots, t_n, K, M, n, k$):
1. Parse sk as $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
2. $G \leftarrow \emptyset$ (this is the set of servers that have sent invalid repair tags)
3. For $k+1 \le i \le n$:
   $t'_i = t_i - \sum_{j=1}^{\ell} g_K(file\_handle||j) g_{K'_i}(file\_handle||i||j)$
4. Decode $(t_1, t_2, \ldots, t_k, t'_{k+1}, \ldots, t'_n)$ using the decoding algorithm of Reed-Solomon codes to obtain message $m = (m_1, m_2, \ldots, m_k)$
5. If the decoding algorithm fails, return $(G, 0)$
6. Otherwise, use $M$ to encode $(m_1, m_2, \ldots, m_k)$, and generate the parity $(m_{k+1}, m_{k+2}, \ldots, m_n)$
7. If none of $m_{k+1}, m_{k+2}, \ldots, m_n$ matches $t'_{k+1}, \ldots, t'_n$, return $(G, 0)$
8. For $1 \le i \le k$: if $m_i \ne t_i$, $G = \{i\} \cup G$
9. For $k+1 \le i \le n$: if $m_i \ne t'_i$, $G = \{i\} \cup G$
10. Return (G, 1)
GenRepairServerCoefficient(sk, $y, k, n, i_1, \ldots, i_k$):
1. Re-generate the information dispersal matrix $M$ by
   running $M \leftarrow$ GenInformationDispersalMatrix($sk, k, n$)
   (a) For $1 \le j \le k$: For $1 \le \alpha \le k$: $A_{j\alpha} = M_{i_j\alpha}$
   (b) Compute matrix $B$ as the inverse of $A$: $B = A^{-1}$
2. If $y \notin \{1, \ldots, k\}$:
   • $C = M_y \times B$   /*$M_y$ is a vector in the $y$-th row of $M$*/
   • For $1 \le j \le k$: $z_{i_j} = C_{1j}$
3. Otherwise:
   • For $1 \le j \le k$: $z_{i_j} = B_{yj}$
4. Return $(z_{i_1}, \ldots, z_{i_k})$
GenRandom(sk, $y, i_1, i_2, \ldots, i_k$):
1. Parse sk as $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
2. $a = g_{K_a}(file\_handle||y||i_1||i_2||\ldots||i_k||1)$
3. $r = g_{K_a}(file\_handle||y||i_1||i_2||\ldots||i_k||2)$
4. For $1 \le j \le k$: $x_j = g_{K_b}(file\_handle||y||i_1||i_2||\ldots||i_k||j)$
5. Return $(a, r, x_1, \ldots, x_k)$
VerifyPartialRepairTag($\mathtt{b}_{i_1}, \mathtt{b}_{i_2}, \ldots, \mathtt{b}_{i_k}, K, t_{i_1}, t_{i_2}, \ldots, t_{i_k}, x_1, x_2, \ldots, x_k$):
1. $G \leftarrow \emptyset$
2. For $1 \le j \le k$:
   If $x_j t_{i_j} \ne \sum_{\alpha=1}^{\ell}(g_K(file\_handle||\alpha))\mathtt{b}_{i_j\alpha}$, $G = G \cup \{i_j\}$
3. Return $G$
RepairOneSegment(sk, $\mathtt{b}'_y, \mathtt{b}''_y, i_1, \ldots, i_k, z_{i_1}, \ldots, z_{i_k}, y, a, r$):
1. Parse sk as $(K_1, K_2, \ldots, K_n, K'_{k+1}, \ldots, K'_n, K_a, K_b)$
2. If $y \in \{1, \ldots, k\}$:
   • For $1 \le j \le \ell$:
     $b_{yj} = a^{-1}(\mathtt{b}'_{yj} - r\mathtt{b}''_{yj}) - z_{i_k} g_{K'_{i_k}}(file\_handle||i_k||j)$
3. Otherwise:
   • For $1 \le j \le \ell$:
     $b_{yj} = a^{-1}(\mathtt{b}'_{yj} - r\mathtt{b}''_{yj}) + g_{K'_y}(file\_handle||y||j)$
4. Return $(\mathtt{b}_{y1}, \ldots, \mathtt{b}_{y\ell})$
```

Figure 4: Components of the RDC-EC scheme (2)

In VerifyAllRepairTag, the client verifies the n repair tags sent back by the n servers based on the secret keys. The client can also localize the corrupted repair tags.

In GenRepairServerCoefficient, the client generates a set of intermediate coefficients, which will be used by the k repair servers to linearly combine their stored segments in order to restore the corrupted segment $\mathtt{b}_y$. The intermediate coefficients are generated as follows: the client first constructs a $k \times k$ square matrix A, in which the elements in the j-th row are copied from the i_j-th row of the information dispersal matrix M, where $1 \le j \le k$. The client then computes B as the inverse of A. If $\mathtt{b}_y$ is a file segment, then the set of intermediate coefficients will be the set of elements in the y-th row of B. If $\mathtt{b}_y$ is a parity segment, then the set of intermediate coefficients will be the set of elements in the first row of $M_y \times B$, where M_y is a $1 \times k$ matrix formed by the elements from the y-th row of M.

In GenRandom, the client generates the random numbers used to mask the intermediate coefficients. To repair the same segment by using the same set of k other segments, the client will use the same set of the random numbers when masking the coefficients.

In VerifyPartialRepairTag, the AS relies on the repair tags $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ to localize the corrupted partial segments, which is feasible because: 1) the client has verified the correctness of $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$, and 2) the repair tag for a partial segment can be derived from the repair tag of the corresponding stored segment. For example, if t_{i_j} is the repair tag for the stored segment $\mathtt{b}_{i_j}$, then xt_{ij} will be the repair tag for its partial segment $x\mathtt{b}_{i_j}$, considering both repair tags are generated based on the same key.

In RepairOneSegment, the client uses intermediate segments $\mathtt{b}'_y$ and $\mathtt{b}''_y$ (received from the servers) to repair corrupted segment $\mathtt{b}_y$.

4.3 Security Analysis for RDC-EC

A unique attack on erasure coding-based storage systems. By knowing the information dispersal matrix M, an adversary can successfully corrupt an erasure coding-based distributed storage system without being detected. The attack can be performed as follows: 1) re-compute the parity by applying M over the original file; 2) compute the secret random numbers embedded into the stored parity by using the stored parity to subtract the parity computed in step 1); 3) replace the original file with a bogus file (with the same size), and apply M over the bogus file, generating the forged parity; 4) embed the secret random numbers into the forged parity, and re-use the server codes as well as the parity for the server codes; 5) upon Challenge, the adversary answers a challenge request relying on the bogus file and the forged parity, and can pass the verification without being detected.

Our RDC-EC scheme can defend against the aforementioned attack, since the adversary cannot learn the information dispersal matrix (Theorem 4.1). In addition, RDC-EC can ensure data recoverability over time under an adversarial setting (Theorem 4.2). Due to space limitations, proofs are provided in [10].

THEOREM 4.1. *In* RDC-EC*, the probability that the adversary can learn the information dispersal matrix M is negligibly small.*

THEOREM 4.2. RDC-EC *can ensure data recoverability over time under an adversarial setting.*

5. Related Work

RDC for the single-server setting. Early remote data checking (RDC) protocols (PDP [4] and PoR [24, 27]) were designed for static data. Later, RDC protocols were designed to allow efficient updates on the outsourced data [5, 20, 32, 29, 36, 12, 11, 35, 28, 30]. Recently, several RDC schemes have been designed for version control systems [21, 14].

RDC for distributed setting. In a distributed RDC, the outsourced data is stored redundantly across multiple servers to achieve a certain reliability level. Distributed RDC protocols include MR-PDP [16], HAIL [7], WWRL [31], RDC-NC [15]. MR-PDP seeks to audit data that is replicated. Like RDC-EC, both HAIL and WWRL are built for erasure coding-based distributed storage systems. However, HAIL is expensive in Repair, because it requires to retrieve all the outsourced data in order to repair a single segment. WWRL is vulnerable to a mobile and Byzantine adversary, because it does not incorporate techniques to address small corruptions, and it only repairs the symbols in a segment being found corrupted during Challenge (spot checking-based). This will lead to a situation that un-checked corrupted symbols remain corrupt at the end of an epoch, and a mobile and Byzantine adversary can possibly corrupt the whole storage system by making a stripe (*i.e.*, the unit of an erasure code) unrecoverable in the following epochs.

New paradigms for RDC. Recently, RDC was applied in several other areas: (a) server-side repair [13], in which a data owner is able to outsource both the data and the management of the data, and thus can remain lightweight during both the challenge and repair phases; (b) proofs of fault tolerance [8], in which a data owner can obtain a proof that the outsourced file is distributed across an expected number of physical storage devices in a single datacenter; and (c) proofs of location (PoL) [6, 33, 22], in which a data owner is offered a guarantee that multiple replicas are stored in data centers located in different or specific geographic locations.

6. Conclusion

We have proposed RDC-EC, an RDC scheme for erasure coding-based distributed storage systems, which functions under an adversarial setting. Unlike previous schemes for this setting, RDC-EC achieves an efficient Repair phase by leveraging server-side repair to shift the burden from the data owner to the storage servers. In the future, we plan to explore mechanisms that can repair multiple data segments more efficiently, and to reduce some of the assumptions related to restoration of the code component at the storage servers.

Acknowledgements. This research was supported by the U.S. National Science Foundation under Grants No. CNS 1054754, CNS 1409523, and DUE 1241976. Bo Chen was also supported in part by NSF grants CNS 1161541, CNS 1318572, CNS 1223239, CCF 0937833, by US ARMY award W911NF-13-1-0142, and by gifts from Northrop Grumman Corporation, Parc/Xerox, and Microsoft Research.

7. References

[1] Microsoft azure. `http://azure.microsoft.com`.

[2] OpenSSL. `http://www.openssl.org/`.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14, June 2011.

[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 Conference on Computer and Communications Security (CCS '07)*, 2007.

[5] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proc. of International ICST Conference on Security and Privacy in Communication Networks (SecureComm '08)*, 2008.

[6] K. Benson, R. Dowsley, and H. Shacham. Do you know where your cloud files are? In *Proc. of ACM Cloud Computing Security Workshop (CCSW '11)*, 2011.

[7] K. Bowers, A. Oprea, and A. Juels. HAIL: A high-availability and integrity layer for cloud storage. In *Proc. of ACM CCS (CCS '09)*, 2009.

[8] K. D. Bowers, M. V. Dijk, A. Juels, A. Oprea, and R. L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Proc. of ACM CCS*, 2011.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 143–157. ACM, 2011.

[10] B. Chen, A. K. Ammula, and R. Curtmola. Towards server-side repair for erasure coding-based distributed storage systems. Technical report, NJIT, December 2014.

[11] B. Chen and R. Curtmola. Poster: Robust dynamic remote data checking for public clouds. In *Proc. of ACM CCS (CCS '12)*, 2012.

[12] B. Chen and R. Curtmola. Robust dynamic provable data possession. In *Proc. of International Workshop on Security and Privacy in Cloud Computing (ICDCS-SPCC '12)*, 2012.

[13] B. Chen and R. Curtmola. Towards self-repairing replication-based storage systems using untrusted clouds. In *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY '13)*, 2013.

[14] B. Chen and R. Curtmola. Auditable version control systems. In *Proc. of the 21th Annual Network and Distrib. System Security Symp. (NDSS '14)*, 2014.

[15] B. Chen, R. Curtmola, G. Ateniese, and R. Burns. Remote data checking for network coding-based distributed storage systems. In *Proc. of ACM Cloud Computing Security Workshop (CCSW '10)*, 2010.

[16] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *Proc. of International Conference on Distributed Computing Systems (ICDCS '08)*, 2008.

[17] A. G. Dimakis, B. Godfrey, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *Proc. of IEEE INFOCOM*, 2007.

[18] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. O. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Inf. Theory*, 56, Sept. 2010.

[19] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *FAST*, volume 9, pages 197–210, 2009.

[20] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. of ACM CCS (CCS '09)*, 2009.

[21] M. Etemad and A. Kupcu. Transparent, distributed, and replicated dynamic provable data possession. In *Proc. of 11th International Conference on Applied Cryptography and Network Security (ACNS '13)*, 2013.

[22] M. Gondree and Z. N. Peterson. Geolocation of data in the cloud. In *Proceedings of ACM CODASPY (CODASPY '13)*, 2013.

[23] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX ATC*, 2012.

[24] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. of ACM Conf. on Computer and Communications Security (CCS '07)*, 2007.

[25] D. A. Patterson, G. Gibson, and R. H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[26] J. S. Plank and K. M. Greenan. Jerasure: A library in C facilitating erasure coding for storage applications – version 2.0. Technical Report UT-EECS-14-721, University of Tennessee, January 2014.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. of ASIACRYPT (ASIACRYPT '08)*, 2008.

[28] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *Proc. of the 20th ACM CCS (CCS '13)*, 2013.

[29] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks. In *Proc. of ACSAC (ACSAC '12)*, 2012.

[30] S. R. Tate, R. Vishwanathan, and L. Everhart. Multi-user dynamic proofs of data possession using trusted hardware. In *Proc. of ACM CODASPY*, 2013.

[31] C. Wang, Q. Wang, K. Ren, and W. Lou. Ensuring data storage security in cloud computing. In *Proc. of IEEE IWQoS (IWQoS '09)*, 2009.

[32] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Trans. on Parallel and Distributed Syst.*, 22(5), May 2011.

[33] G. J. Watson, R. Safavi-Naini, M. Alimomeni, M. E. Locasto, and S. Narayan. LoSt: location based storage. In *Proc. of ACM CCSW (CCSW '12)*, 2012.

[34] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitiative comparison. In *Proc. of IPTPS*, 2002.

[35] Y. Zhang and M. Blanton. Efficient dynamic provable possession of remote data via balanced update trees. In *Proc. of ACM ASIACCS (ASIACCS '13)*, 2013.

[36] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proc. of ACM Conf. on Data and Application Security and Privacy (CODASPY '11)*, 2011.