Remote Data Checking for Network Coding-based Distributed Storage Systems

Bo Chen, Reza Curtmola
Department of Computer Science
New Jersey Institute of Technology
{bc47,crix}@njit.edu

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

ABSTRACT

Remote Data Checking (RDC) is a technique by which clients can establish that data outsourced at untrusted servers remains intact over time. RDC is useful as a *prevention* tool, allowing clients to periodically check if data has been damaged, and as a *repair* tool whenever damage has been detected. Initially proposed in the context of a single server, RDC was later extended to verify data integrity in distributed storage systems that rely on replication and on erasure coding to store data redundantly at multiple servers. Recently, a technique was proposed to add redundancy based on *network coding*, which offers interesting tradeoffs because of its remarkably low communication overhead to repair corrupt servers.

Unlike previous work on RDC which focused on minimizing the costs of the prevention phase, we take a holistic look and initiate the investigation of RDC schemes for distributed systems that rely on network coding to minimize the combined costs of both the prevention and repair phases. We propose RDC-NC, a novel secure and efficient RDC scheme for network coding-based distributed storage systems. RDC-NC mitigates new attacks that stem from the underlying principle of network coding. The scheme is able to preserve in an adversarial setting the minimal communication overhead of the repair component achieved by network coding in a benign setting. We implement our scheme and experimentally show that it is computationally inexpensive for both clients and servers.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.4 [**Coding and Information Theory**]: Error Control Codes

General Terms

Security, Reliability, Performance

Keywords

cloud storage, remote data integrity checking, dynamic provable data possession, robustness, small corruption

1. INTRODUCTION

Remote data checking (RDC) has been shown to be a valuable technique by which a client (acting as a verifier) can efficiently establish that data stored at an untrusted server remains intact over time [2, 15, 22]. This kind of assurance is essential to ensure long-term reliability of data outsourced at data centers or at cloud storage providers. When used with a single server, the most valuable use of remote data checking lies within its prevention capability: The verifier can periodically check data possession at the server and can thus detect data corruption. However, once corruption is detected, the single server setting does not necessarily allow data recovery. Thus, remote data checking has to be complemented with storing the data redundantly at multiple servers. In this way, the verifier can use remote data checking with each server and, upon detecting data corruption at any of the servers, it can use the remaining healthy servers to restore the desired level of redundancy by storing data on a new server.

The main approaches to introduce redundancy in distributed storage systems are through *replication*, *erasure coding*, and more recently through *network coding* [8, 9]. The basic principle of data replication is to store multiple copies of the data at different storage servers, whereas in erasure coding the original data is encoded into fragments which are stored across multiple storage servers. In network coding, the coded blocks stored across servers are computed as linear combinations of the original data blocks.

Network coding for distributed storage systems and application scenarios. Network coding for storage [8, 9] provides unusual performance properties well suited to deep archival stores which are characterized by a *read-rarely* workload. The parameters of network coding make reading data more expensive than data maintenance. Similar with erasure coding, network coding can be used to redundantly encode a file into fragments and store these fragments at n servers so that the file can be recovered (and read) from any k servers. However, network coding provides a *significant advantage* over erasure coding when coded fragments are lost due to server failures and need to be reconstructed in order to maintain the same level of reliability: a new coded fragment can be constructed with optimally minimum communication cost by contacting some of the healthy servers (the repair bandwidth can be made as low as the repaired fragment). This is in sharp contrast with conventional erasure codes, such as Reed-Solomon codes [19] which must rebuild the entire file prior to recovering from data loss. Recent results in network coding for storage have established that the maintenance bandwidth can be reduced by orders of magnitude compared to standard erasure codes.

The proposals for using network coding in storage have one *drawback* though: the code is not systematic; it does not embed the input as part of the encoded output. Small portions of the file cannot be read without reconstructing the entire file. Online storage systems do not use network coding, because they prefer to optimize performance for read (the common operation). They use systematic codes to support sub-file access to data. Network-coding for storage really only makes sense for systems in which data repair occurs much more often than read.

Regulatory storage, data escrow, and deep archival applications present read-rarely workloads that match well the performance properties of network coding. These applications preserve data for future access with few objects being accessed during any period of time. Many of these applications do not require sub-file access; they retrieve files in their entirety. Auditing presents several examples, including keeping business records for seven years in accordance with Sarbanes-Oxley and keeping back tax returns for five years. Only those records that are audited or amended ever need to be accessed, but retaining all data is a legal or regulatory requirement. Medical records are equally relevant. The Johns Hopkins University Medical Image Archive retains all MRI, CAT-scan, and X-ray images collected in the hospitals in a central repository of more than 6 PB. A small fraction of images are ever accessed for historical tracking of patients or to examine outcomes of similar cases. Preservation systems for the storage of old books, manuscripts, data sets also present a read-rarely workload. Furthermore, standards for archival storage [1] represent data as an indivisible package and do not support subfile access. In applications, the size of the data and the infrequency of reads dictate that the performance of storage maintenance, re-encoding to mitigate data loss from device or system failures, dominates the performance requirements of read.

A holistic approach for long-term reliability. To ensure long-term data reliability in a distributed storage system, after data is redundantly stored at multiple servers, we can loosely classify the actions of a verifier into two components: *prevention* and *repair*. In the prevention component, the verifier uses remote data checking protocols to ensure the integrity of the data at the storage servers. In the repair component, which is invoked when data corruption is detected at any of the servers, the client uses data from the healthy servers to restore the desired redundancy level. Over the lifetime of a storage system, the prevention and repair components will alternate. Eventually, there will also be a *retrieval* component, in which the client recovers the original data (although this happens rarely for archival storage systems).

In this paper, we take a holistic approach and propose novel techniques to minimize the combined costs of the prevention and repair components. Previous work on remote data checking has focused exclusively on minimizing the cost of the prevention component (*e.g.*, replication [7] and erasure coding [24, 4] based approaches). However, in the distributed storage settings we consider, the cost of the repair component is significant because over a long period of time servers fail and data needs to be re-distributed on new servers. Our work builds on recent work in coding for distributed storage [8, 9], which leverages network coding to achieve a remarkable reduction in the communication overhead of the repair component compared to an erasure coding-based approach. However, this work was proposed for a benign setting. In essence, we seek to preserve in an adversarial setting the minimal communication overhead of the repair component when using network coding. The main challenge towards achieving this goal stems from the very nature of network coding: In the repair phase, the client must ensure the correctness of the coding operations performed by servers, without having access to the original data. At the same time, the client storage should remain small and constant over time, to conform with the notion of outsourced storage.

The need for remote data checking in distributed storage systems. Archival storage also requires introspection and data checking to ensure that data are being preserved and are retrievable. Since data are rarely read, it is inadequate to only check the correctness and integrity of data on retrieval. Storage errors, from device failures, torn writes [17], latent errors [21], and mismanagement may damage data undetectably. Also, storage providers may desire to hide data loss incidents in an attempt to preserve their reputation or to delete data maliciously to reduce expenses [2]. Deep archival applications employ data centers, cloud storage, and peer-to-peer storage systems [18] in which the management of data resides with a third party: not with the owner of the data. This furthers the need for the data owner to check the preservation status of stored data to audit whether the third party fulfills its obligation to preserve data.

The performance properties of remote data checking protocols, such as provable data possession [2] and proofs of retrieval [15], also conform to read-rarely workloads. These protocols allow an auditor to guarantee that data are intact on storage and retrievable using a constant amount of client metadata, constant amount of network traffic, and (most importantly) by reading a constant number of file fragments [2]. Large archival data sets make it prohibitive to read every byte periodically. Remote data checking protocols sample stored data to achieve probabilistic guarantees. When combined with error correcting codes, the guarantees can reach confidence of 10^{-10} for practical parameters [6]. Error correcting codes ensure that small amounts of data corruption do no damage, because the corrupted data may be recovered by the code, and that large amounts of data corruption are easily detected, because they must corrupt many blocks of data to overcome the redundancy.

The combination of remote data checking and network coding makes it possible to manage a read-rarely archive with a minimum amount of I/O. Specifically, one can detect damage to data and recover from data using I/O sub-linear in the file size: a constant amount I/O per file to detect damage and I/O in proportion to the amount of damage to repair the file.

Contributions. In this paper, we take a holistic look at remote data checking and consider distributed storage systems that minimize the combined costs of both the prevention and repair components. We propose novel RDC schemes that build on recent work in coding for distributed storage systems, which leverages network coding to achieve a remarkable reduction in the communication overhead of the repair component compared to erasure coding-based approaches. To the best of our knowledge, we are the first to consider remote data checking for network coding-based distributed storage systems which rely on untrusted servers. In this context, we identify new attacks and propose a novel RDC scheme for network coding-based storage systems. Specifically, our paper makes the following contributions:

– We take a holistic approach and propose a novel remote data checking scheme for distributed storage systems that minimize the combined costs of both prevention and repair components. Network coding-based systems exhibit minimal communication overhead of the repair component in a benign setting. Our scheme preserves in an adversarial setting the communication overhead advantage of network coding-based over erasure coding-based systems.

– Our RDC schemes overcome challenges that are unique to network coding-based distributed storage systems (details in Sec. 3):

- unlike single-server RDC schemes [2, 15, 22], which can only detect if corruption exists *somewhere* in the data, practical considerations require RDC schemes to *localize* the faulty servers in a multiple-server setting.
- unlike erasure coding-based distributed storage systems, network coding-based systems lack a fixed file layout, which makes it challenging for RDC schemes to maintain constant client storage over time.
- compared to erasure coding-based distributed systems, network coding-based systems are vulnerable to additional attacks. We identify the *replay* and *pollution* attacks. The proposed RDC schemes successfully mitigate these attacks. To render replay attacks harmless, we use a simple but effective solution: the network coding coefficients are stored encrypted on the server; moreover, the client is the one that chooses the coding coefficients and enforces their use. To prevent pollution attacks, we use an additional repair verification tag,

	Replication (MR-PDP [7])	Erasure Coding (HAIL [4])	Network Coding (RDC-NC)
Total server storage	$O(n\vert\mathtt{F}\vert)$	$O(\frac{n\vert\mathtt{F}\vert}{k})$	$O(\frac{2n\vert\mathtt{F}\vert}{k+1})$
Communication (repair phase)	$O(\vert\mathtt{F}\vert)$	$O(\vert\mathtt{F}\vert)$	$O(\frac{2\vert\mathtt{F}\vert}{k+1})$
Network overhead factor (repair phase)	1	k	1
Server computation (repair phase)	$O(1)$	$O(1)$	$O(1)$

Table 1: Parameters of various RDC schemes. We assume that our scheme RDC-NC uses an MBR code, under the additional constraint to minimize the total server storage (more details in Sec. 3.1.5). For the repair phase, we describe the costs for the case when one storage server fails.

which allows the client to check that a server combines its blocks correctly during the repair phase.

– We provide guidelines on how to apply network coding in the context of a distributed storage system that relies on untrusted servers (Sec. 3.1.5). We evaluate through experiments the performance of the proposed RDC schemes.

Solution overview. Table 1 compares our scheme (RDC-NC) with previous RDC schemes. The underlying substrate for adding redundancy in RDC-NC is based on network coding, whereas in previous work it is based on replication and erasure-coding.

To ensure the security of the prevention component, we adapt RDC techniques used in the single server setting [22]. We present a scheme in which only the data owner can check data possession (*i.e.*, it is privately verifiable). However, our scheme can be extended using the techniques in [2, 22] to achieve public verifiability for the prevention phase (*i.e.*, anyone, not just the data owner, can challenge the server to prove data possession).

For the security of the repair component, our solution ensures that the data provided by contributing servers is valid and preserves the amount of desired redundancy in the system. This will ultimately ensure that the original data can be recovered after an arbitrarily many repairs. The *replay attacks* we identify may lead to a reduction in data redundancy, similar to the entropy attacks identified by Jiang *et al.* [14] in a network setting, in which intermediate nodes forward non-innovative packets. However, the solution of Jiang *et al.* relies on checking if a new coded packet is linearly independent with all previously coded packets. In our distributed storage setting, their solution cannot preserve the minimal communication overhead of the repair component. The *pollution attacks* we identify are similar with the pollution attacks that may occur when network coding is used to improve throughput of communication over a network. A line of work on signatures for network coding [3, 11] ensures that intermediate nodes perform correctly the encoding operations. However, our storage setting is different because the client is the one that chooses the coding coefficients and enforces their use by the servers. Moreover, the solution in [11] leads to an increase in the size of the coded blocks after each encoding operation and cannot be used in a long-term storage setting where the number of repair operations is unbounded.

2. BACKGROUND ON DISTRIBUTED STORAGE SYSTEMS

We give an overview of the main approaches proposed in distributed storage systems to store data redundantly across multiple storage servers: replication-based, erasure coding-based, and network coding-based. These are effective in a non-adversarial setting, where only benign faults may occur. For each, we outline the storage cost to store data redundantly and the network cost to restore the desired level of redundancy when a server fails. We also formulate the *data recovery condition*, which captures the amount of corruption that can be tolerated without affecting the ability to recover the original data. These approaches are illustrated in Fig. 1.

We consider a file $\mathtt{F}$ that needs to be stored redundantly (we denote the size of $\mathtt{F}$ by $|\mathtt{F}|$). To express the network overhead of the repair component, we define the *network overhead factor* as the ratio between the amount of data that needs to be retrieved (from healthy servers) to the amount of data that is created to be stored on a new server. This will be our primary metric to measure the communication cost of the repair component.

2.1 Replication

Replication is the simplest form of redundancy and many storage systems have adopted it. The client stores one file replica at each of ℓ servers. Thus, the original file can be recovered from any of the ℓ servers. The storage cost is $\ell|\mathtt{F}|$ across all servers. Upon detecting corruption of a replica, the client can use any one of the healthy replicas to create a new replica. As part of the repair component, in order to create a new replica of size $|\mathtt{F}|$, the client needs to retrieve a replica of size $|\mathtt{F}|$. Thus, the network overhead factor is 1.

Data recovery condition: *The original file can be recovered as long as at least one of the ℓ replicas is not corrupted.*

2.2 Erasure Coding

In erasure coding, given a file $\mathtt{F}$ of k blocks, the client uses an (n, k) maximum distance separable erasure code to create n coded blocks out of the original k file blocks, and stores them at n servers (one coded block per server). Thus, the original file can be recovered from any k out of the n servers. Whenever the client detects corruption of one of the coded blocks, it can use the remaining healthy blocks to regenerate the corrupted coded block. The storage cost is $|\mathtt{F}|\frac{n}{k}$ across all servers ($\frac{|\mathtt{F}|}{k}$ per server). This is optimal in terms of redundancy-reliability storage tradeoff[1]. However, compared with the replication-based solution, erasure coding has a higher network overhead cost for the repair component: To create one new coded block, the client has to first reconstruct the entire file (*i.e.*, retrieve k coded blocks), thus incurring a network overhead factor of k.

Data recovery condition: *The original file can be recovered as long as at least k out of the n coded blocks are not corrupted.*

2.3 Network Coding for Distributed Storage

Recent work in coding for distributed storage [8, 9] has shown that the k network overhead factor for the repair component is not unavoidable (as it was commonly believed). Given a file represented by m input blocks, $\bar{\mathtt{b}}_1, \bar{\mathtt{b}}_2, \ldots, \bar{\mathtt{b}}_m$, the client uses network coding to generate coded blocks as linear combinations of the original m file blocks. Each input block $\bar{\mathtt{b}}_i$ can be viewed as a column vector: $\bar{\mathtt{b}}_i = (b_{i1}, b_{i2}, \ldots, b_{iu})$, where b_{ij} are elements in a finite field $GF(2^w)$ and are referred to as *symbols*.

Given a coding coefficient vector $(x_1, \ldots, x_m)$, in which x_i are

[1] Compared with replication, erasure coding achieves a reliability level that is an order of magnitude higher for the same redundancy level [25].

chosen at random from $GF(2^w)$, a coded block $\bar{\mathtt{c}}$ is computed as a linear combination of the input blocks:

$$\bar{\mathtt{c}} = \sum_{i=1}^{m} x_i \bar{\mathtt{b}}_i, \text{ where all algebraic operations are over } GF(2^w).$$

The linear combinations of the symbols in the input blocks are performed over a finite field using randomly chosen coefficients. Thus, a coded block has the same size as an original file block and can also be viewed as a column vector $\bar{\mathtt{c}} = (c_1, \ldots, c_u)$. It has been shown [12, 13] that if the coding coefficients are chosen at random from a large enough field (*i.e.*, at least $GF(2^8)$), then the original file can be recovered from m coded blocks by solving a system of m equations (because the m coded blocks will be linearly independent with high probability).

These coded blocks are then stored at servers, with each server storing α' bits[2], which comprises of $\alpha = \alpha'/|\mathtt{B}|$ coded blocks, where $|\mathtt{B}| = |\mathtt{F}|/m$ denotes the size of a block (both original and coded). Thus, $\alpha = \alpha' m/|\mathtt{F}|$.

To achieve a similar reliability level as in erasure coding, the client stores data on n servers such that any k servers can be used to recover the original file with high probability. This means that any k servers will collectively store at least m coded blocks.

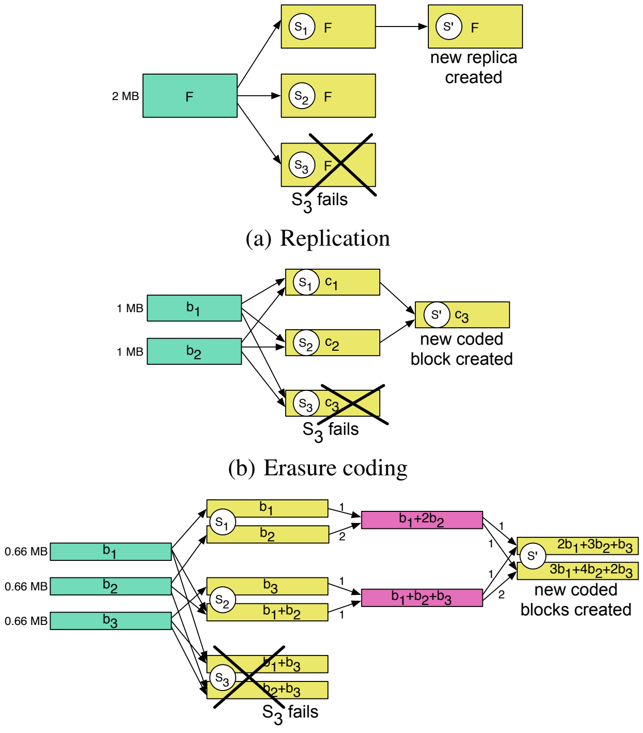
When the client detects corruption at one of the storage servers, it contacts ℓ healthy servers and retrieves from each server β' bits (which comprises of $\beta = \beta'/|\mathtt{B}| = \beta' m/|\mathtt{F}|$ coded blocks, obtained as linear combinations of the blocks stored by the server). The client then further linearly combines the retrieved blocks to generate α coded blocks to be stored at a new server. Unlike in the erasure coding-based approach, the client does not have to reconstruct the entire file in order to generate coded blocks for a new server; instead, the coded blocks retrieved from healthy servers contain enough novel information to generate new coded blocks. The network overhead factor is thus less than k.

The storage cost is $n\alpha'$ bits across all servers (α' bits per server). The network overhead of the repair component is $\gamma' = \ell\beta'$ bits, so the network overhead factor is $\frac{\gamma'}{\alpha'}$. There is a tradeoff between the storage cost and the repair network overhead cost [9]. In short, for every tuple $(n, k, \ell, \alpha', \gamma')$, there exists a family of solutions which has two extremal points on the optimal tradeoff curve:

– One extremal point uses the pair $(\alpha', \gamma') = \left(\frac{|\mathtt{F}|}{k}, \frac{|\mathtt{F}|\ell}{k(\ell-k+1)}\right)$ to minimize the storage cost on the servers. It is referred to as a minimum storage regenerating (MSR) code. The storage cost per server is $|\mathtt{F}|/k$, same as in the erasure coding-based approach[3], but this approach has a network overhead factor of $\frac{\ell}{\ell-k+1}$ and outperforms erasure coding in terms of network cost of the repair component whenever $\ell > k$.

– The other extremal point minimizes the network overhead of the repair component by using the pair $(\alpha', \gamma') = \left(\frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}, \frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}\right)$. It is referred to as a minimum bandwidth regenerating (MBR) code. Remarkably, it incurs a network overhead factor of 1, the same as a replication-based approach. The tradeoff is that this point requires each server to store (slightly) more data than in erasure coding.

Data recovery condition: *The original file can be recovered as long as at least k out of the n servers collectively store at least m coded blocks which are linearly independent combinations of the original m file blocks.*



(a) Replication

(b) Erasure coding

(c) Network coding

Figure 1: Example of various approaches for redundantly storing a file $\mathtt{F}$ of 2 MB:
(a) In *replication*, copies of the file are stored at three servers, S_1, S_2, S_3. When the replica at S_3 gets corrupted, the client uses the non-corrupted replica at S_1 to create a new replica of size 2 MB. The client retrieves 2 MB in order to generate a new replica of size 2 MB, so the network overhead factor is 1.
(b) In *erasure coding*, the original file has two 1 MB blocks ($\mathtt{b}_1, \mathtt{b}_2$) and is encoded into three blocks ($\mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3$), using a $(3, 2)$ erasure code (so that $\mathtt{F}$ can be reconstructed from any two coded blocks). Each coded block is stored at a different server. When $\mathtt{c}_3$ gets corrupted, the client first retrieves $\mathtt{c}_1$ and $\mathtt{c}_2$ to reconstruct $\mathtt{F}$ and then regenerates the coded block $\mathtt{c}_3$. The client retrieves 2 MB in order to generate a new block of size 1 MB, so the network overhead factor is 2.
(c) In *network coding*, the original file has three 0.66 MB blocks and the client computes coded blocks as linear combinations of the original blocks. Two such coded blocks are stored on each of three storage servers. Note that this choice of parameters respects the guarantees of a $(3, 2)$ erasure code (*i.e.*, any two servers can be used to recover $\mathtt{F}$, because they will have at least three linearly independent equations, which allows to reconstruct the original blocks $\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3$). When the data at S_3 gets corrupted, the client uses the remaining two servers to create two new blocks: The client first retrieves one block from each healthy server (obtained as a linear combination of the server's blocks), and then further mixes these blocks (using linear combinations) to obtain two new coded blocks which are stored at a new server. The numbers on the arrows represent the coefficients used for the linear combinations. The client retrieves 1.33 MB in order to generate a new coded block of size 1.33 MB, so the network overhead factor is 1.

An Example. We illustrate the three approaches in Fig. 1.

[2] For each coded block, the coding coefficients are also stored. This assumption is implicit in any network coding-based system and for simplicity we do not add the coefficients to the storage cost as their size is usually negligible compared to the actual coded data.

[3] Indeed, this extremal point provides the same reliability-redundancy performance with erasure coding.

3. REMOTE DATA CHECKING SCHEMES FOR NETWORK CODING-BASED DISTRIBUTED STORAGE SYSTEMS

In this section, we present *remote data checking* (RDC) schemes for distributed storage systems based on network coding. Once the client detects the failure of a server, it needs to take measures to ensure the data recovery condition is maintained. Recently, RDC schemes were proposed for replication-based [7] and erasure coding-based [24, 4] distributed storage systems. To the best of our knowledge, RDC was not considered for network coding-based distributed storage systems. In this paper, we seek to achieve remote data checking for distributed systems that use network coding to store data redundantly at multiple storage servers.

Challenges. We summarize the challenges that need to be overcome when moving from a single server to a multiple server setting. We also point out the challenges that arise in a network coding-based system over an erasure coding-based system.

– *The need to localize the faulty servers in a multiple-server setting.* In a single-server setting, after using remote data checking to detect file corruption, the client takes the decision to repair the *entire* file and is not concerned with detecting which file blocks are corrupted and which are still valid. In distributed storage however, when a fault occurs, the client cannot afford to repair data on all servers. Instead, the client has to identify the faulty servers and use the remaining servers to restore the faulty servers to a good state. Since preserving constant client storage over time and localizing faulty servers are opposite goals, techniques used for dynamic RDC [10] in a single-server setting are not directly applicable.

– *Lack of a fixed file layout in network coding-based distributed storage vs. erasure coding-based storage.* In erasure-coding, when a faulty block is detected, the repair phase reconstructs the *same exact* block. Whereas in network coding, repair computes a *new block*, which is different from the original blocks. Thus, it is more challenging to preserve the ability to check the integrity of newly coded blocks, while maintaining constant storage on the client.

– *Additional attacks.* A network coding-based system is vulnerable to attacks that are specific to network coding and do not occur in erasure coding-based systems. In a *replay attack*, the adversary attempts to reuse old coded blocks in order to reduce the redundancy on the storage servers to the point that the original data becomes unrecoverable. In a *pollution attack*, corrupted servers use correct data to avoid detection in the challenge phase, but provide corrupted data for coding new blocks in the repair phase. The client must ensure that servers combine correctly their blocks during repair, without having access to the original blocks.

System and adversarial model. Initially, the client stores data redundantly across a set of n storage servers, $S_1, S_2, \dots, S_n$.

We adopt an adversarial model similar to the one in HAIL [4]. We assume a *mobile adversary* that can behave arbitrarily (*i.e.*, exhibits Byzantine behavior) and can corrupt any (and potentially all) of the servers over the system lifetime. However, the adversary can corrupt at most $n-k$ 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.

In every epoch, the adversary may choose a new set of $n-k$ servers and corrupt both the code and the storage component on these servers. However, at the end of each epoch, we assume that the code component of each server is restored to a correct state[4]. Although the code component is restored, 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 and cause the original data to 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 faulty servers, it uses the redundancy at the remaining healthy servers to repair data at faulty servers.

An epoch consists of two phases:

1. 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 performs checks of data possession with the servers. As a result, the client may detect servers with corrupted storage (*i.e.*, *faulty* servers).
2. A *repair phase* that contains two sub-phases and is triggered only if corruption is detected during 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 attacker during an epoch is at most $n-k$ (*i.e.*, $b_1 + b_2 \leq n-k$).

The structure of an epoch is similar with the one in [4], with one 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.

3.1 RDC for Network Coding

In this section, we present our main RDC-NC scheme. To facilitate the exposition, we gradually introduce a series of challenges and our approaches to overcome them, leading to the main RDC-NC scheme in Sec. 3.1.4.

We consider remote data checking schemes for a storage system that relies on network coding to store and repair data as described in Sec. 2.3. The client chooses a set of parameters $(n, k, \ell, \alpha', \gamma')$ (in Sec. 3.1.5 we give guidelines on how to choose the parameters in a setting where the storage servers are untrusted). The file $\mathtt{F}$ is split into m blocks, $\bar{\mathtt{b}}_1, \dots, \bar{\mathtt{b}}_m$. The client computes and stores $\alpha = \frac{\alpha' m}{|\mathtt{F}|}$ coded blocks at each of the n servers (*i.e.*, server S_i stores coded blocks $\bar{\mathtt{c}}_{i1}, \dots, \bar{\mathtt{c}}_{i\alpha}$). A coded block is computed as a linear combination of the original m file blocks.

3.1.1 Can Existing RDC Schemes be Extended?

We start by examining challenges that arise when designing RDC schemes for a network coding-based distributed storage system. These stem from the underlying operating principle of network coding, which computes new coded blocks as linear combinations of existing blocks in the repair phase. We first focus on the challenge phase, followed by the repair phase.

The challenge phase. Single-server RDC schemes compute verification metadata, which is stored together with the data and facilitates data integrity checks [2, 22]. Such schemes can be extended to a multiple-server setting if we regard each coded block as a collection of *segments* and the client computes a challenge verification tag for each segment. A single-server RDC scheme based on spot checking [2, 22] can then be used to check integrity of each of the α blocks stored by each of the n servers.

[4] From a practical point of view, this is equivalent to removal of malware by reinstalling software.

This approach must ensure that server S_i stores the blocks that it is supposed to store (*i.e.*, blocks $\bar{c}_{i1}, \ldots, \bar{c}_{i\alpha}$). Note that a direct application of a single-server RDC scheme does not achieve this guarantee, as a malicious server S_i could simply store the blocks and tags of another (honest) server and successfully pass the integrity challenges. This would reduce the overall redundancy across servers and will eventually lead to a state where the file becomes unrecoverable, without the client's knowledge. To prevent this attack, the index of a block must be embedded into the verification tags associated with the segments of that block.

In a distributed storage system that uses erasure coding to store a file redundantly across multiple servers, each of the n storage servers is assigned one erasure-coded block (*i.e.*, server S_i is assigned erasure-coded block i). For erasure coding, the layout of the encoded file is fixed and is known to the client (because the client knows the parity matrix used for the erasure code). As a result, when coded block i is found corrupted, the *same exact* block i will be reconstructed by the repair phase. Thus, it is straightforward to embed the index of a block into challenge verification tags, as the client can use the same index i to challenge possession of the i-th block regardless of how many repair operations have occurred.

However, when the storage system relies on network coding (as opposed to erasure coding), one complication arises because the layout of the coded file is not fixed anymore. As servers fail, the client does not reconstruct the same blocks on the failed servers (like in the erasure coding-based solution). Instead, the client retrieves new coded blocks from the healthy servers and recombines them using randomly chosen coefficients to obtain other new coded blocks. Thus, it becomes challenging to maintain constant storage on the client and, at the same time, verify that each server stores the blocks it is supposed to store.

The repair phase. A malicious server may store the correct blocks it is supposed to store, and may act honestly in the challenge phase. However, the server may act maliciously in the repair phase, when it is asked to contribute coded blocks during the process of reconstructing blocks after a server failure. If the server does not combine its blocks correctly and contributes corrupted blocks, the corrupted blocks will lead to further corruption in the system as they are being combined with blocks from other servers. This *pollution* attack is possible because the client does not have access to the original blocks in order to check if the encoding operation was performed correctly by the server. Our RDC-NC scheme in Sec. 3.1.4 prevents pollution attacks in the repair phase by using a *repair verification tag* (different from challenge verification tags).

3.1.2 How to Maintain Constant Client Storage?

The intuition behind our solution to maintain constant storage on the client is that we assign logical identifiers to the coded blocks stored at each server and embed these identifiers into the challenge verification tags. Each server stores α coded blocks, thus the n servers collectively store $n\alpha$ coded blocks. We assign logical identifiers to these $n\alpha$ coded blocks, in the form "$i.j$", where i is the server number and j is the block number stored at server i. For example, in Fig. 1(c), the blocks stored at servers S_1 have identifiers "1.1" and "1.2", and the blocks stored at S_3 have identifiers "3.1" and "3.2". Note that server S_1 cannot pass integrity challenges by using blocks with identifiers "2.1" or "2.2", which are supposed to be stored on S_2.

When S_3 fails, the new coded blocks computed by the client to be stored on S' maintain the same identifiers "3.1" and "3.2". This identifier is embedded into the challenge verification tags for the segments of each coded block. When the client wants to challenge the blocks stored on S', it can regenerate the logical identifiers "3.1" and "3.2" (thus the client can maintain constant storage).

We place no restriction on which server stores a block with a certain logical identifier, as long as each server keeps track of the logical identifier of the blocks it stores. Thus, we allow coded blocks to freely migrate among storage servers. The only assumption we make is the existence of a discovery service which can locate the server which stores the block with a given logical identifier "$i.j$".

3.1.3 Replay Attacks

One concern with using logical identifiers is that a malicious server can reuse previously coded blocks with the same logical identifier (even old coded blocks that were previously stored on failed servers) in order to successfully pass the integrity verification challenge. The data recovery condition could be broken if the malicious server is able to find and reuse old coded blocks such that they are linearly dependent with other coded blocks that are currently stored across servers. In essence, this is a *replay attack* that could potentially lead to breaking the data recovery condition. In Appendix B, we give a concrete replay attack example for a configuration similar with the one in Fig. 1.

Simple defense against replay attacks. The client stores an additional *version* information in the challenge verification tags. The version acts as a counter that starts from 0 and is incremented each time the blocks on a server are recreated due to server failure. There is only one version counter for all the blocks at a server (repairing a faulty server involves regenerating all the blocks on that server). The client needs to store locally the latest value of the counter for each of the n servers. In addition to the usual integrity check, the client also checks that its current counter value is embedded in the tags used by the servers to pass the check, which prevents a server from passing the check by using an old version of a block with the same logical identifier. The storage cost for the client now becomes $O(n)$, which may be acceptable in practical settings where n (the number of storage servers) is small. However, $O(n)$ client storage does not conform with our notion of outsourced storage and, from an asymptotic point of view, a more efficient solution is desirable.

Mitigating replay attacks. While the example in Appendix B shows that successful replay attacks are possible, the example is somewhat artificial in that the coding coefficients used in the repair phase are specifically chosen in order to break the data recovery condition. In fact, replaying old coded blocks is only harmful if it introduces additional linear dependencies among the coded blocks stored at the servers. Otherwise, replay attacks are not harmful, because the underlying principle that ensures the validity of network coding is preserved.

We mitigate replay attacks by using a combination of methods:
– In traditional network coding, the coding coefficients are stored in plaintext at the servers together with their corresponding coded blocks. Instead, we require the client to store the coding coefficients in encrypted format, which will prevent the adversary from knowing how the original blocks were combined to obtain the coded block. Thus, even if the adversary corrupts servers, its ability to execute harmful replay attacks is negligible, as it does not have the necessary knowledge to target certain blocks on certain servers or to know which old coded blocks to replay.
– In the repair phase, the client chooses the random coefficients that should be used by the servers and enforces their use by the servers. This is described in our RDC-NC scheme in Sec 3.1.4.

Theorem 3.1 shows that by encrypting the coding coefficients, a malicious server's ability to execute a harmful replay attack becomes negligible.

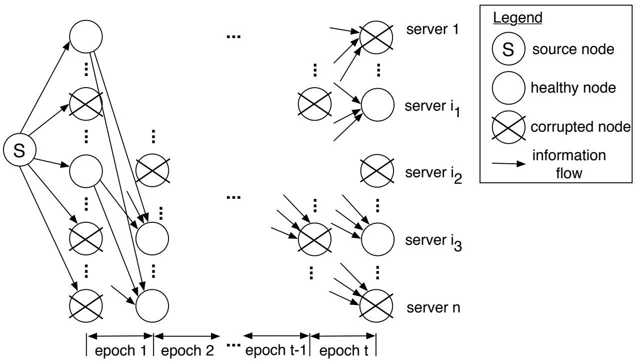


Figure 2: An illustration of the information flow graph after t epochs. A node in this graph represents the storage at a specific server in a particular epoch. The source node S has the original file, which is then encoded using network coding and stored at n servers. In each epoch, the data on at most $n-k$ servers can be corrupted (due to either benign or adversarial faults). At the end of each epoch, the servers with corrupted data are detected and repaired using data from k healthy servers. An information flow arrow incoming into a node in epoch i means that the node is repaired at the end of epoch i using data from healthy nodes.

THEOREM 3.1. *We consider a network coding-based distributed storage system in which any k out of n servers can recover a file with high probability. Let P_1 be the probability to recover the file in a setting where the coding coefficients are stored in plaintext and the storage servers are not malicious (*i.e. *a benign setting). Let P_2 be the probability to recover the file in a setting where the coding coefficients are stored encrypted and the adversary can corrupt storage servers and execute replay attacks. Then, $|P_1 - P_2| \leq \epsilon$, where $\epsilon \to 0$ (*i.e.*, these probabilities are negligibly close).*

PROOF. (sketch) In Fig. 2, we use a representation of the distributed storage system similar with the information flow graph used in [8, 9] to describe how encoded data is stored at storage nodes and how data is transmitted for the purpose of repairing faulty storage nodes and for recovering the original data. In each epoch the data on at most $n-k$ servers can be corrupted due to benign faults or due to the attacker. At the end of each epoch, the servers with corrupted data are detected and repaired using data from k healthy servers.

We first consider the benign setting, where only non-adversarial faults may occur. The system guarantees that the original file can be recovered from any k out of the n nodes with high probability. This guarantee holds for any epoch, after the repair phase. Equivalently, there are $\binom{n}{k}$ receivers [13] that can recover the file (*i.e.*, receiver 1 can recover the file from servers $1,\ldots,k$, receiver 2 can recover the file from servers $1,\ldots,k-1,k+1$, etc).

We then consider a setting in which data may be corrupted due to both non-adversarial or adversarial faults (*i.e.*, servers may fail due to benign faults or the adversary may target to corrupt data on specific servers). Corrupted data is detected and repaired in each epoch. However, the attacker records all data prior to corruption and accumulates all old coded blocks. Let N_t denote the number of all nodes in the information flow graph after t epochs, except for the source node (this includes both healthy and corrupted nodes). Thus, in epoch $t+1$, the attacker can access data from N_t nodes to execute the replay attack. We distinguish between two cases:
– *Unencrypted coefficients*. The coding coefficients are stored in plaintext together with the coded data. We now show that, at the end of epoch t, the system must guarantee that any k nodes out of N_t can recover the file, under the condition that the k nodes belong to different servers (since the logical identifier ensures that a node can only be used for a specific server). If, at the end of epoch t, there exists at least one set of k nodes out of N_t which belong to different servers and which do not have enough information to recover the file, then the attacker can use this set to cause permanent damage (we refer to this one set as the "bad" set). Then, over the course of at most $\lceil \frac{k}{n-k} \rceil$ epochs ($n-k$ servers per epoch), the attacker can execute replay attacks and gradually replace current data on the k servers corresponding to the nodes in the "bad" set with old data from the nodes in the "bad" set. The replay attacks will not be detected. Then, in the next epoch, the attacker corrupts data on the other $n-k$ servers, causing permanent file damage, since the k servers corresponding to the "bad" set do not have enough information to recover the file.

Equivalently, there are $\binom{N_t}{k} - f(t)$ receivers that should be able to recover the file, where $f(t)$ represents the number of receiver that are connected to nodes that belong to the same server. We have $(t-1)(\binom{n}{k}-1)+\binom{n}{k} \leq \binom{N_t}{k} - f(t) \leq t^n\binom{n}{k}$. According to Theorem 1 in [13], for a fixed field size, a system based on random linear network coding has an upper bound on the number of receivers it can support. Thus, since $\binom{N_t}{k} - f(t)$ grows unbounded as t grows, the system cannot guarantee that the original file can be recovered.
– *Encrypted coefficients*. The coding coefficients are stored encrypted together with the coded data. Since the attacker has no knowledge of the coding coefficients, it has no better strategy than picking at random data it has stored from nodes that were corrupted in the past, and using this data to replace data currently stored on servers. Note that replay attacks remain undetectable only if old data from server i is replayed on the same server i, because of the logical identifiers embedded in the challenge verification tags. This means that, unlike in the case of unencrypted coefficients, the system should only guarantee that at the end of epoch t the file can be recovered from any k out of n nodes. But this is the same guarantee that is already achieved by the system in the benign case! Intuitively, the guarantee is preserved because, even if there exists a "bad" set of k nodes (belonging to k different servers), the attacker cannot identify these nodes (as the coefficients are encrypted) and can only pick nodes at random for the replay attack. □

Thus, we conclude that when coding coefficients are encrypted, the replay attack is a negligible threat. As a result, the client does not need to take other explicit countermeasures to mitigate the replay attack, besides encrypting the coefficients. For the remainder of this section, we will give a solution in which the coding coefficients are encrypted before being stored on the server.

*3.1.4 Remote Data Checking for Network Coding (*RDC-NC*)*

We are now ready to present the network coding-based RDC scheme (RDC-NC) that provides defense against both direct data corruption attacks and replay attacks, and is able to maintain constant client storage. This scheme is the main result of the paper.

Recall that the file $\mathtt{F}$ is split into m blocks, $\bar{\mathtt{b}}_1,\ldots,\bar{\mathtt{b}}_m$. The client computes and stores $\alpha = \frac{\alpha' m}{|\mathtt{F}|}$ coded blocks at each of n servers (*i.e.*, server i stores coded blocks $\bar{\mathtt{c}}_{i1},\ldots,\bar{\mathtt{c}}_{i\alpha}$). We use the notation $\bar{\mathtt{c}}_{ij}$ to refer to the j-th coded block stored by the i-th server). A coded block is computed as a linear combination of the original m file blocks.

We use two independent logical representations of file blocks, for different purposes:

– For the purpose of checking data possession (in the challenge phase), a block (either original or coded) is viewed as an ordered collection of s segments. For example a coded block $\bar{\mathtt{c}}_{ij} =$

Recall from Sec. 2.3 that $\alpha = \alpha' m/|\mathtt{F}|$ and $\beta = \beta' m/|\mathtt{F}|$. We construct a network coding-based RDC scheme in three phases, Setup, Challenge, and Repair. Let $f : \{0,1\}^* \times \{0,1\}^\kappa \to GF(p)$ be a PRF and let (KGen, Enc, Dec) be a semantically secure encryption scheme. We work over the field $GF(p)$ of integers modulo p, where p is a large prime (at least 80 bits).

Setup: The client generates the secret key $\text{sk} = (K_{prf1}, K_{prf2}, K_{prf3}, K_{prf4}, K_{enc})$, where each of these five keys is chosen at random from $\{0,1\}^\kappa$. The client then executes: For $1 \le i \le n$:

(a) Generate a value δ using f and K_{prf1}: $\delta = f_{K_{prf1}}(i)$ (the δ value will be used for generating the challenge tags)
Generate u values λ_1, ..., λ_u using f and K_{prf2}: $\lambda_k = f_{K_{prf2}}(i||k)$, with $1 \le k \le u$ (the λ values will be used for generating the repair tag)
For $1 \le j \le \alpha$: // generate coded blocks and metadata to be stored at server S_i

- run $(\bar{\mathtt{c}}_{ij}, z_{ij1}, \ldots, z_{ijm}, \mathtt{t}_{ij1}, \ldots, \mathtt{t}_{ijs}, \mathtt{T}_{ij}) \leftarrow$ GenBlockAndMetadata$(\bar{\mathtt{b}}_1, \ldots, \bar{\mathtt{b}}_m, \text{“}i.j\text{”}, \delta, \lambda_1, \ldots, \lambda_u, \text{sk})$
- For $1 \le k \le m$: $\epsilon_{ijk} = \mathsf{Enc}_{K_{enc}}(z_{ijk})$ //encrypt coefficients $z_{ij1}, \ldots, z_{ijm}$

(b) Send $\bar{\mathtt{c}}_{ij}, \epsilon_{ij1}, \ldots, \epsilon_{ijm}, \mathtt{t}_{ij1}, \ldots, \mathtt{t}_{ijs}, \mathtt{T}_{ij}$ to server S_i for storage, with $1 \le j \le \alpha$

The client may now delete the file $\mathtt{F}$ and stores only the secret key sk.

Challenge: For each of the n servers, the client checks possession of each of the α coded blocks stored at a server (by using spot checking of segments for each coded block). In this process, each server uses its stored blocks and the corresponding challenge tags to prove data possession. A detailed description of this phase is provided in App. A.

Repair: Assume that in the challenge phase C has identified a faulty server whose blocks have logical identifiers “$y.1$”, ..., “$y.\alpha$”. The client C contacts ℓ healthy servers $S_{i_1}, \ldots, S_{i_\ell}$ and asks each one of them to generate a new coded block (step 1). The client further combines these ℓ coded blocks to generate α new coded blocks and metadata (step 2), and then stores them on a new server S' (step 3).

1. For each $i \in \{i_1, \ldots, i_\ell\}$:

(a) C generates a set of coefficients $(x_1, \ldots, x_\alpha)$, where $x_k \overset{R}{\leftarrow} GF(p)$, with $1 \le k \le \alpha$

(b) C asks server S_i to provide a new coded block and a proof of correct encoding using the coefficients $(x_1, \ldots, x_\alpha)$

(c) server S_i runs $(\bar{\mathtt{a}}_i, \tau_i) \leftarrow$ GenRepairBlock$(\bar{\mathtt{c}}_{i1}, \ldots, \bar{\mathtt{c}}_{i\alpha}, x_1, \ldots, x_\alpha, \mathtt{T}_{i1}, \ldots, \mathtt{T}_{i\alpha})$ and sends $(\bar{\mathtt{a}}_i, \tau_i, \epsilon_{i11}, \ldots, \epsilon_{i1m}, \epsilon_{i21}, \ldots, \epsilon_{i2m}, \ldots, \epsilon_{i\alpha 1}, \ldots, \epsilon_{i\alpha m})$ to C

(d) C decrypts the encrypted coefficients ϵ to recover coefficients $z_{i11}, \ldots, z_{i1m}, z_{i21}, \ldots, z_{i2m}, \ldots, z_{i\alpha 1}, \ldots, z_{i\alpha m}$

(e) C re-generates u values λ_1, ..., $\lambda_u \in GF(p)$ using f keyed with K_{prf2}: $\lambda_k = f_{K_{prf2}}(i||k)$, with $1 \le k \le u$

(f) If $\tau_i \neq \sum_{j=1}^{\alpha} x_j f_{K_{prf4}}(i||j||z_{ij1}||z_{ij2}||\ldots||z_{ijm}) + \sum_{k=1}^{u} \lambda_k a_{ik} \bmod p$, then C declares S_i faulty (here, $a_{i1}, \ldots, a_{iu}$ are the symbols of block $\bar{\mathtt{a}}_i$)

2. Generate a value δ using f and K_{prf1}: $\delta = f_{K_{prf1}}(y)$ (the δ value will be used for generating the challenge tags)
Generate u values λ_1, ..., λ_u using f and K_{prf2}: $\lambda_k = f_{K_{prf2}}(y||k)$, with $1 \le k \le u$ (the λ values will be used for generating the repair tag)
For $1 \le j \le \alpha$:

- C runs $(\bar{\mathtt{c}}_{yj}, z_{yj1}, \ldots, z_{yjm}, \mathtt{t}_{yj1}, \ldots, \mathtt{t}_{yjs}, \mathtt{T}_{yj}) \leftarrow$ GenBlockAndMetadata$(\bar{\mathtt{a}}_{i_1}, \ldots, \bar{\mathtt{a}}_{i_\ell}, \text{“}y.j\text{”}, \delta, \lambda_1, \ldots, \lambda_u, \text{sk})$
- For $1 \le k \le m$: $\epsilon_{yjk} = \mathsf{Enc}_{K_{enc}}(z_{yjk})$ //encrypt coefficients $z_{yj1}, \ldots, z_{yjm}$

3. Client C sends $\bar{\mathtt{c}}_{yj}, \epsilon_{yj1}, \ldots, \epsilon_{yjm}, \mathtt{t}_{yj1}, \ldots, \mathtt{t}_{yjs}, \mathtt{T}_{yj}$ for storage to server S', with $1 \le j \le \alpha$

Figure 3: RDC-NC: a network coding-based RDC scheme

GenBlockAndMetadata$(\bar{\mathtt{b}}_1, \ldots, \bar{\mathtt{b}}_m, \text{“}i.j\text{”}, \delta, \lambda_1, \ldots, \lambda_u, \text{sk})$:
(run by the client to generate a coded block with logical identifier “$i.j$” and its associated metadata)

1. Parse sk as $(K_{prf1}, K_{prf2}, K_{prf3}, K_{prf4}, K_{enc})$
2. For $1 \le k \le m$: randomly generate coding coefficients $z_k \overset{R}{\leftarrow} GF(p)$
3. Compute coded block $\bar{\mathtt{c}}_{ij} = \sum_{k=1}^{m} z_k \bar{\mathtt{b}}_k$ //the symbols in the vector $\bar{\mathtt{c}}_{ij}$ are elements in $GF(p)$
4. View block $\bar{\mathtt{c}}_{ij}$ as an ordered collection of s segments $\bar{\mathtt{c}}_{ij} = (\mathtt{c}_{ij1}, \ldots, \mathtt{c}_{ijs})$, where each segment contains one symbol from $GF(p)$, and compute a *challenge tag* for each segment:
For $1 \le k \le s$: $\mathtt{t}_{ijk} = f_{K_{prf3}}(i||j||k||z_1||z_2||\ldots||z_m) + \delta \mathtt{c}_{ijk} \bmod p$
5. View $\bar{\mathtt{c}}_{ij}$ as a column vector of u symbols $\bar{\mathtt{c}}_{ij} = (c_{ij1}, \ldots, c_{iju})$, with $c_{ijk} \in GF(p)$, and compute a *repair tag* for block $\bar{\mathtt{c}}_{ij}$:
$\mathtt{T}_{ij} = f_{K_{prf4}}(i||j||z_1||z_2||\ldots||z_m) + \sum_{k=1}^{u} \lambda_k c_{ijk} \bmod p$
6. Return $(\bar{\mathtt{c}}_{ij}, z_1, \ldots, z_m, \mathtt{t}_{ij1}, \ldots, \mathtt{t}_{ijs}, \mathtt{T}_{ij})$

GenRepairBlock$(\bar{\mathtt{c}}_{i1}, \ldots, \bar{\mathtt{c}}_{ik}, x_{i1}, \ldots, x_{ik}, \mathtt{T}_{i1}, \ldots, \mathtt{T}_{ik})$:

1. Compute $\bar{\mathtt{a}}_i = \sum_{j=1}^{k} x_{ij} \bar{\mathtt{c}}_{ij}$ (here the symbols a_{ir} of block $\bar{\mathtt{a}}_i$ are computed as $a_{ir} = \sum_{j=1}^{k} x_{ij} c_{ijr} \bmod p$, for $1 \le r \le u$)
2. Compute a proof of correct encoding $\tau_i = \sum_{j=1}^{k} x_{ij} \mathtt{T}_{ij} \bmod p$
3. Return $(\bar{\mathtt{a}}_i, \tau_i)$

Figure 4: Components of the RDC-NC scheme

$(\mathtt{c}_{ij1}, \ldots, \mathtt{c}_{ijs})$, where each segment $\mathtt{c}_{ijk}$ is a contiguous portion of the block $\bar{\mathtt{c}}_{ij}$ (in fact, each segment contains one symbol).
– For the purpose of network coding, a block (either original or coded) is viewed as a column vector of u symbols (as described in Sec. 2.3). For example, a coded block $\bar{\mathtt{c}}_{ij} = (c_{ij1}, \ldots, c_{iju})$, where $c_{ijk} \in GF(p)$[5].

Consequently, we use two types of verification tags. To check data possession (in the challenge phase) we use challenge verification tags (in short *challenge tags*), and to ensure the security of the repair phase we use repair verification tags (in short *repair tags*). There is one challenge tag for each segment in a block, and one repair tag for each block. From a notational point of view, we use $\mathtt{t}$ (lowercase) for challenge tags and $\mathtt{T}$ (uppercase) for repair tags.

To detect direct data corruption attacks, the client performs a spot checking-based challenge similar as in POR [22] and PDP [2] to check the integrity of each network coded block stored by each of the n servers. The challenge tag for a segment in a coded block binds the data in the segment with the block's logical identifier and also with the coefficient vector that was used to obtain that block. Thus, the client implicitly verifies that the server cannot use segments from a block with a different logical identifier to pass the challenge, and also that the coefficient vector retrieved by the client corresponds to the block used by the server to pass the challenge. If a faulty server is found in the challenge phase, the client uses the remaining healthy servers to construct new coded blocks in the repair phase and stores them on a new server.

The details of RDC-NC scheme are presented in Figures 3 and 4. We rely on a construction of a message authentication code which, as indicated by Bowers *et al.* [4], has been proposed as a combination of universal hashing with a PRF [26, 16, 20, 23]. Shacham and Waters [22] use a similar construction. We adapt this construction to our network coding-based distributed storage setting.

[5] Note that, unlike the description for network coding in Sec. 2.3 which uses symbols from $GF(2^w)$, we use symbols from the finite field $GF(p)$ over the integers modulo p, where p is a prime such that $GF(p)$ and $GF(2^w)$ have about the same order (*e.g.*, when $w = 8$, p is 257). Based on a similar analysis as in [11], the successful decoding probability for network coding under $GF(2^w)$ and under $GF(p)$ is similar.

The Setup **phase.** The client first generates secret key material. It then generates the coded blocks and the metadata to be stored on each of the n servers. For each server, the client calls GenBlockAndMetadata α times in order to generate the coded blocks, the coding coefficients, the challenge tags corresponding to segments in each coded block, and the repair tag corresponding to each coded block. The coding coefficients and the tags are then stored on the server, together with their corresponding blocks (the coefficients are first encrypted by the client).

In GenBlockAndMetadata, the client picks random coefficients from $GF(p)$ and uses them to compute a new coded block $\bar{\mathsf{c}}_{ij}$ (steps 2 and 3). For each segment in the coded block, the client computes a challenge tag which is stored at the server and will be used by the server in the Challenge phase to prove data possession. For each segment in the coded block $\bar{\mathsf{c}}_{ij}$, the client embeds into the challenge tags of that segment the coefficient vector used to obtain $\bar{\mathsf{c}}_{ij}$ from the original file blocks (step 4). For example, in Fig. 1(c), the second block stored at server S_1 has been computed using the coefficient vector $[0, 1, 0]$ (and its logical identifier is "1.2"). Thus, the challenge tag for the k-th segment in this block is[6]:

$$f_{K_{prf3}}(\underbrace{1||2}_{\text{block logical identifier}} \; ||k|| \; \underbrace{0||1||0}_{\text{coefficient vector}}) + \delta\mathsf{b}_{2k} \bmod p.$$

For each coded block, the client also computes a repair verification tag (step 5), which will be used in the Repair phase to ensure that the server used the correct blocks and the coefficients provided by the client to generate new coded blocks.

The Challenge **phase.** For this phase, we rely on the scheme in [22], adapted as described in Sections 3.1.1 and 3.1.2 (in short, the challenge tags include the logical identifier of the block, the index of the segment, and the coding coefficients used to obtain that block). Note that we rely on the scheme in [22] that allows private verifiability (*i.e.*, only the data owner can check possession of the data). However, RDC schemes that allow public verifiability [2, 22] could also be adapted following a similar strategy.

The Repair **phase.** The client contacts ℓ healthy servers $S_{i_1}, \ldots, S_{i_\ell}$ and asks each one of them to generate a new coded block (step 1)[7]. The client further combines these ℓ coded blocks to generate α new coded blocks and metadata (step 2), and then stores them on a new server S' (step 3). As part of step 1, for each contacted server, the client chooses random coefficients from $GF(p)$ that should be used by the server to generate the new coded block (step 1(a)). Each contacted server, based on its α stored blocks and on the coefficients provided by the client, runs GenRepairBlock (step 1(c)) to compute a new coded block $\bar{\mathsf{a}}_i$ (GenRepairBlock, step 1) and a proof of correct encoding τ_i (GenRepairBlock, step 2); the server then sends these to the client, together with the encrypted coefficients corresponding to the blocks used to compute the new coded block. The client decrypts the coefficients (step 1(d)), re-generates the λ values used to compute the repair tags (step 1(e)), and then checks whether the encoding was done correctly by the server (step 1(f)). As a result, the client is ensured that the server has computed the new coded block by using the correct blocks and the coefficients supplied by the client.

A tradeoff between storage and communication. In the RDC-NC scheme described in Figures 3 and 4, each segment contains only one symbol (*i.e.*, an element from $GF(p)$) and is accompanied by a challenge tag of equal length. However, the scheme can be modified to use a similar strategy as in [22]: each segment can contain r symbols, which reduces the server storage overhead by a factor of r and increases the client-server communication overhead of the challenge phase by a factor of r.

Protection against small data corruption. A spot checking mechanism for the challenge phase is only effective in detecting "large" data corruption [2, 15]. To protect against "small" data corruption, we can combine the RDC-NC scheme with error correction codes such as a "server code" [5, 4] or a method to add "robustness" [6]. The client applies a server code on the network coded blocks before computing the challenge and repair tags. In the repair phase, a server does not include the server code portion when computing new coded blocks. Instead, the client computes the server code for the new coded blocks. We leave as future work the design of schemes in which the server code portion can be computed together with the rest of the block using network coding.

Security analysis. We now restate the data recovery condition for a network coding-based system and then give a theorem that states the sufficiency of this condition to ensure file recoverability.

Data recovery condition: *In any given epoch, the original data can be recovered as long as at least k out of the n servers collectively store at least m coded blocks which are linearly independent combinations of the original m file blocks.*

THEOREM 3.2. *The data recovery condition is a sufficient condition to ensure data recoverability in the* RDC-NC *scheme augmented with protection against small data corruption.*

PROOF. (sketch) In its initial state (*i.e.*, right after Setup), the system based on RDC-NC guarantees that the original file can be recovered from any k out of the n servers with high probability. We want to show that the RDC-NC scheme preserves this guarantee throughout its lifetime, thus ensuring file recoverability.

In any given epoch, the adversary can corrupt at most $n - k$ servers. The adversary may split the corruptions between direct data corruptions and replay attacks. Faulty servers affected by direct data corruptions are detected by the integrity checks in the challenge phase, or by the correct encoding check in the repair phase. The client uses the remaining healthy servers (at least k remain healthy) to regenerate new coded blocks to be stored on new servers. Protection against small data corruption is provided by the server code layer. From Theorem 3.1, replay attacks against RDC-NC are not harmful, and they do not increase the attacker's advantage in breaking the data recovery condition. The check in Repair, step 1(f), ensures protection against pollution attacks. Thus, at the end of the epoch, the system is restored to a state equivalent to its initial state, in which the file can be recovered from any k out of the n servers. □

3.1.5 Guidelines for Choosing Parameters for RDC-NC

For a benign setting, the network coding-based substrate of our RDC-NC scheme is characterized by a tuple of parameters $(n, k, \alpha, m, \ell, \beta)$. Compared to a benign setting, an adversarial setting imposes additional constraints for choosing these parameters. In this section, we provide guidelines for choosing the parameters, subject to two constraints: (a) up to $n - k$ servers can be corrupted in each epoch, and (b) minimize the total server storage. Not all these parameters are independent, and we will see that fixing the values for some of the parameters will determine the value of the remaining parameters.

Based on the desired reliability level (which is a function of perceived fault rate of the storage environment), the client picks the values for n and k. We focus on MBR codes, which are charac-

[6] The input to the PRF are encoded as fixed length strings.

[7] For ease of exposition, we use $\beta = 1$ (*i.e.*, each server produces only one new coded block), but this can be easily generalized to any value of β.

case	n	k	α	m	l	β
1	10	3	3	6	3	1
2	12	3	3	6	3	1
3	10	5	5	15	5	1

Table 2: Experimental test cases

terized by the pair $(\alpha', \gamma') = \left(\frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}, \frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}\right)$ and which minimize the network overhead factor of the repair phase (*i.e.*, $\gamma'/\alpha' = 1$). Although we give guidelines on choosing parameters for MBR codes, our RDC-NC scheme is general and can be applied to any parameter tuple.

After fixing n, k, and the use of an MBR code, we study the choice of the remaining parameters. Before the system is initialized, the client also needs to decide the values of m and α, so that it can compute the initial coded blocks that will be stored at servers.

From $\alpha' = \frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}$, $m = |\mathtt{F}|/|\mathtt{B}|$, and $\alpha' = |\mathtt{B}|\alpha$, we get:

$$\alpha = \frac{2\ell m}{2k\ell - k^2 + k} \qquad (1)$$

Even though we have fixed a point characterized by a minimal network overhead factor of 1, different values of the parameters α, m, and ℓ will result in different storage overheads at the servers. We now show that by choosing $\ell = k$, we minimize the total storage across the n servers (recall that ℓ is the number of healthy servers that are contacted by the client in the repair phase). We need to provision the system for the worst case, in which the adversary corrupts $n - k$ servers in some epoch. In this case, the maximum number of servers that remain healthy in that epoch is k. Thus, we need to set $\ell \leq k$. Minimizing the total storage across the n servers means we need to minimize the quantity $n\alpha' = n|\mathtt{F}|\frac{\alpha}{m}$. Since n is fixed, for a given file size $|\mathtt{F}|$, we should minimize $\frac{\alpha}{m}$. From Eq. (1), $\frac{\alpha}{m} = \frac{2\ell}{2k\ell-k^2+k} = \frac{2}{2k-\frac{k^2-k}{\ell}}$, which is minimized when ℓ is maximized. Thus, we need to set $\ell = k$.

From $\gamma' = \frac{2|\mathtt{F}|\ell}{2k\ell-k^2+k}$ and from $\gamma' = \ell\beta\frac{|\mathtt{F}|}{m}$, we get:

$$m = \frac{\beta(2k\ell - k^2 + k)}{2} = \frac{\beta(k^2 + k)}{2} \qquad (2)$$

From $\alpha' = \gamma'$, $\alpha' = \alpha|\mathtt{B}|$, and $\gamma' = \ell\beta|\mathtt{B}|$ we get:

$$\alpha = \ell\beta = k\beta \qquad (3)$$

Thus, after fixing n and k, under the constraints of achieving a minimal network overhead factor of 1 and minimizing the total storage across the n servers, it suffices to choose the β parameter in order to determine the values of m and α.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the computational performance of our RDC-NC scheme (the analysis of its communication and storage overhead can be found in Table 1 of Sec. 1).

Implementation issues: Working over $GF(p)$ rather than $GF(2^w)$. When working over $GF(2^w)$, the symbols used in network coding have exactly w bits. One implementation complication arises when working over $GF(p)$, where p is a prime. In theory, the symbols should be represented using $\lceil \lg(p) \rceil$ bits. For example, when $p = 257$, 9 bits should be used to represent a symbol. However, when encoding the file in the pre-processing phase, we cannot treat a chunk of 9 consecutive bits in the original file as a symbol, because the value represented by that chunk may be larger (or equal) than 257 (since with 9 bits we can represent values up to 511). Instead, before the encoding step in the pre-processing phase, we run an additional step in which we read 8-bit chunks and write them as 9-bit chunks (this leads to an increase in storage of 12.5%). This ensures that every 9-bit chunk has a value less than 257 and is thus a valid symbol.

Experimental Setup. All experiments were conducted on an Intel Core 2 Duo system with two CPUs (each running at $3.0GHz$, with a $6144KB$ cache), $1.333GHz$ frontside bus, $4GB$ RAM and a Hitachi $HDP725032GLA360$ $360GB$ hard disk with $ext3$ file system. The system runs Ubuntu 9.10, kernel version 2.6.31-14-$generic$. The implementation uses OpenSSL version 1.0.0. We work over the finite field $GF(p)$ of integers modulo p, where p is an 80-bit prime and each segment for the challenge phase contains one 80-bit symbol.

When evaluating the performance of RDC-NC, we are only concerned with pre-processing (in the setup phase) and the repair phase (the performance of the challenge phase was evaluated in [2]).

We choose to evaluate three cases, as shown in Table 2. They satisfy the condition of MBR codes with minimal server storage (see Sec. 3.1.5). We use $\beta = 1$.

4.1 Pre-Processing Phase Results

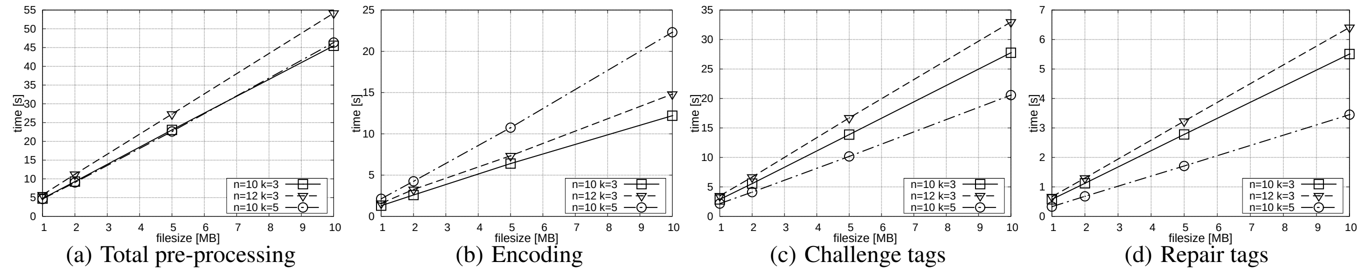
The client pre-processes the file before outsourcing it. Specifically, the client: (a) encodes the m-block file using network coding over $GF(p)$, generating $n\alpha$ coded blocks (α blocks for each server), (b) computes the challenge tags, and (c) computes the repair tags. Figure 5 shows the computational cost of client pre-processing and its various components for different file sizes under the three test cases. Note that the amount of data that needs to be pre-processed can be significantly larger than the original file size. For example, for test case 1 ($n = 10, \alpha = 3, m = 6$), if $|\mathtt{F}| = 10$MB then the client has to pre-process $|\mathtt{F}|\alpha n/m$ = 50MB. We can see that these computational costs are all increasing linearly with the file size. The cost of generating challenge and repair tags is determined by the total amount of pre-processed data $n\alpha'$, which can be further expressed as $\frac{2n*|\mathtt{F}|}{k+1}$ ($\alpha' = \frac{2*|\mathtt{F}|}{k+1}$, inferred from Sec. 3.1.5). Thus, for fixed $|\mathtt{F}|$, the cost for generating challenge and repair tags increases with n and decreases with k.

4.2 Repair Phase Results

We assume that the client needs to repair one server.

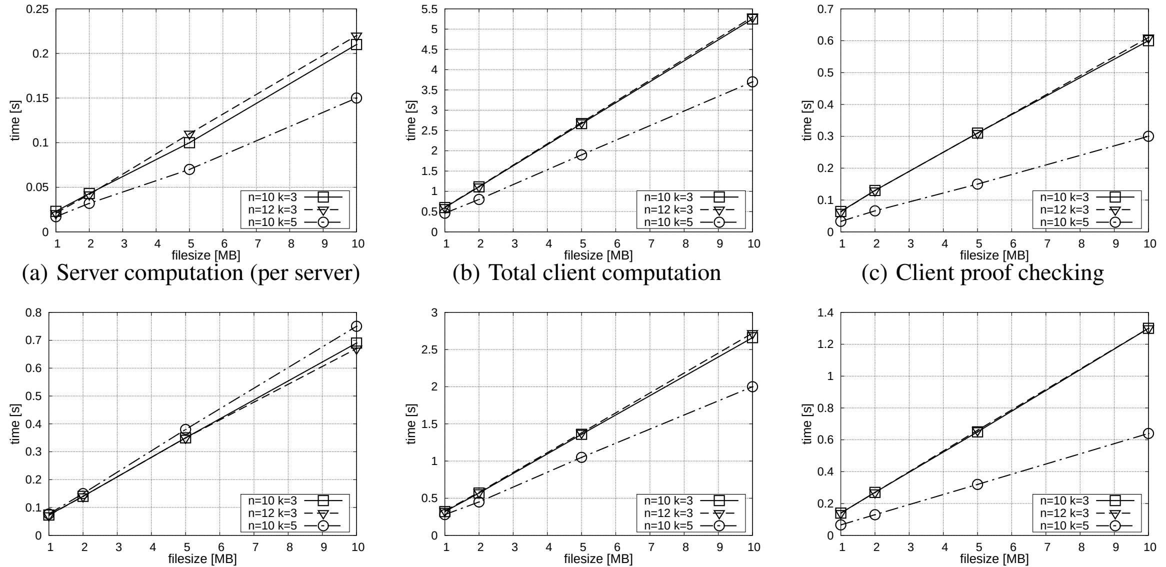
Server computation. In the repair phase, the client retrieves blocks from ℓ servers. Every server generates β new blocks, together with the aggregation of the tags using the coefficients sent by client. Figure 6(a) shows the per-server computational cost under the three test cases. The computational cost increases linearly with the file size for all test cases. We observe that the per server computation varies by k, rather than by n; specifically, for fixed file size, when k increases, per server computation decreases. This can be explained as follows. Per server computation contains two components, encoding and aggregating the tags, and is dominated by the encoding component. The cost of encoding is $O(\frac{\beta\alpha|\mathtt{F}|}{m})$, which can be further expressed as $O(\frac{2*|\mathtt{F}|}{k+1})$ (cf. Sec. 3.1.5, with $\beta = 1$). Thus, for fixed $|\mathtt{F}|$, per server computation is only determined by k in a monotonically decreasing way.

Client computation. After getting β blocks from each of the ℓ servers, the client checks the proof sent by each server. The client then generates α new coded blocks from these $\beta\ell$ blocks (using random network coding over $GF(p)$), together with the challenge tags and repair tags. Figures 6(b)-6(f) show the total client computational cost to repair one server and the costs of its various components. These figures show that the client computational cost is linear to the file size and varies by k, rather than by n. The storage at



(a) Total pre-processing (b) Encoding (c) Challenge tags (d) Repair tags

Figure 5: Computational cost for client pre-processing and its various components (to pre-process data for n servers).



(a) Server computation (per server) (b) Total client computation (c) Client proof checking

(d) Client network coding new blocks (e) Client generating new challenge tags (f) Client generating new repair tags

Figure 6: The computational cost of the repair phase: (a) server cost, (b)-(f) client cost to repair one server.

one server, $\frac{2*|\mathsf{F}|}{k+1}$, determines the cost of proof checking, generating challenge tags and repair tags, thus can explain why computational costs of these components are decreasing with k for fixed file size. However, the cost of encoding in the repair phase is $O(\frac{\alpha\beta\ell|\mathsf{F}|}{m})$, *i.e.*, $O(\frac{|\mathsf{F}|}{1+1/k})$ (cf. Sec. 3.1.5, with $\beta = 1$), thus, encoding cost increases with k for fixed file size.

5. CONCLUSION

In this paper, we propose a secure and efficient RDC scheme for network coding-based distributed storage systems that rely on untrusted servers. Our RDC-NC scheme can be used to ensure data remains intact when faced with data corruption, replay, and pollution attacks. The performance evaluation shows that RDC-NC is inexpensive for both clients and servers.

6. REFERENCES

[1] Reference model for an open archival information system (OAIS), 2001. Consultative Committee for Space Data Systems.

[2] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., and Song, D. Provable data possession at untrusted stores. In *Proc. of ACM CCS* (2007).

[3] Boneh, D., Freeman, D., Katz, J., and Waters, B. Signing a linear subspace: Signature schemes for network coding. In *Proc. PKC '09*.

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

[5] Bowers, K. D., Juels, A., and Oprea, A. Proofs of retrievability: Theory and implementation. In *Proc. of the 2009 ACM Workshop on Cloud Computing Security (CCSW '09)* (2009).

[6] Curtmola, R., Khan, O., and Burns, R. Robust remote data checking. In *Proc. of ACM StorageSS* (2008).

[7] Curtmola, R., Khan, O., Burns, R., and Ateniese, G. MR-PDP: Multiple-replica provable data possession. In *Proc. of ICDCS* (2008).

[8] Dimakis, A. G., Godfrey, B., Wainwright, M. J., and Ramchandran, K. Network coding for distributed storage systems. In *INFOCOM* (2007).

[9] Dimakis, A. G., Godfrey, P. B., Wu, Y., Wainwright, M. O., and Ramchandran, K. Network coding for distributed storage systems. *IEEE Transactions on Information Theory* (2010).

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

[11] Gennaro, R., Katz, J., Krawczyk, H., and Rabin, T. Secure network coding over the integers. In *Proc. of PKC '10* (2010).

[12] Ho, T., Koetter, R., Medard, M., Karger, D. R., and Effros, M. The benefits of coding over routing in a randomized setting. In *Proc. of IEEE International Symposium on Information Theory (ISIT)* (2003).

[13] Ho, T., Medard, M., Koetter, R., Karger, D. R., Effros, M., Shi, J., and Leong, B. A random linear network coding approach to multicast. *IEEE Trans. Inform. Theory 52*, 10 (2006), 4413–4430.

[14] Jiang, Y., Fan, Y., Shena, X., and Lin, C. A self-adaptive probabilistic packet filtering scheme against entropy attacks in network coding. *Elsevier Computer Networks* (August 2009).

[15] JUELS, A., AND KALISKI, B. S. PORs: Proofs of retrievability for large files. In *Proc. of ACM CCS* (2007).

[16] KRAWCZYK, H. LFSR-based hashing and authentication. In *Crypto '94*.

[17] KRIOUKOV, A., BAIRAVASUNDARAM, L. N., GOODSON, G. R., SRINIVASAN, K., THELEN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Parity lost and parity regained. In *Proc. of FAST'08* (2008).

[18] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T., ROSENTHAL, D., BAKER, M., AND MULIADI, Y. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[19] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[20] ROGAWAY, P. Bucket hashing and its application to fast message authentication. In *Proc. of CRYPTO '95* (1995).

[21] SCHROEDER, B., DAMOURAS, S., AND GILL, P. Understanding latent sector errors and how to protect against them. In *Proc. of FAST'10* (2010).

[22] SHACHAM, H., AND WATERS, B. Compact proofs of retrievability. In *Proc. of Asiacrypt 2008* (2008).

[23] SHOUP, V. On fast and provably secure message authentication based on universal hashing. In *Proc. of CRYPTO '96* (1996).

[24] WANG, C., WANG, Q., REN, K., AND LOU, W. Ensuring data storage security in cloud computing. In *Proc. of IWQoS* (2009).

[25] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: a quantitiative comparison. In *Proc. of IPTPS* (2002).

[26] WEGMAN, M. N., AND CARTER, J. L. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences 22*, 3 (1981), 265 – 279.

APPENDIX

A. THE Challenge PHASE OF THE RDC-NC SCHEME

The Challenge phase is described in Figures 7 and 8. For a more detailed description of the context for the Challenge phase, we refer the reader to [22, 2]. The communication cost can be reduced by performing an aggregate check for all α blocks stored at a server. In addition to Q, the client sends to S_i α random numbers, $a_1, ..., a_\alpha$ (which can also be generated pseudo-randomly). Instead of responding with $(\mathtt{t}_{ij}, \rho_{ij})$ for each block, S_i aggregates $\mathtt{t}_{ij}$ by computing $\mathtt{t}_i = \sum_{j=1}^{\alpha} a_j \mathtt{t}_{ij} \bmod p$ and aggregates ρ_{ij} by computing $\rho_i = \sum_{j=1}^{\alpha} a_j \rho_{ij} \bmod p$, then sends back $(\mathtt{t}_i, \rho_i)$. The client then checks whether $t_i = \sum_{j=1}^{\alpha} a_j \sum_{(k,\nu_k)\in Q} \nu_k f_{K_{prf3}}(i||j||k||z_{ij1}||\dots||z_{ijm}) + \delta\rho_i \bmod p$.

Challenge: For each of the n servers, the client checks possession of each of the α coded blocks stored at each server (by using spot checking of segments for each coded block). In this process, each server uses its stored blocks and the corresponding challenge tags to prove data possession.

For $1 \le i \le n$:

(a) C randomly generates r pairs (k, ν_k), where $k \overset{R}{\leftarrow} [1, s]$ and $\nu_k \overset{R}{\leftarrow} GF(p)$. Let the query Q be the r-element set $\{(k, \nu_k)\}$. C sends Q to S.
(The (k, ν_k) pairs could also be pseudo-randomly generated – this would reduce the server-client communication – but for simplicity here we generate them at random.)

(b) For $1 \le j \le \alpha$: S_i runs $(\mathtt{t}_{ij}, \rho_{ij}) \leftarrow$ GenProofPossession$(Q, \bar{\mathtt{c}}_{ij}, \mathtt{t}_{ij1}, \dots, \mathtt{t}_{ijs})$. S_i sends to C the proofs of possession $(\mathtt{t}_{ij}, \rho_{ij})$ and the encrypted coding coefficients $(\epsilon_{i11}, \dots, \epsilon_{i1m}, \epsilon_{i21}, \dots, \epsilon_{i2m}, \dots, \epsilon_{i\alpha 1}, \dots, \epsilon_{i\alpha m})$ corresponding to the α coded blocks at S_i.

(c) For $1 \le j \le \alpha$:
// C checks the validity of the proof of possession $(\mathtt{t}_{ij}, \rho_{ij})$
– Decrypt the encrypted coefficients: $z_{ijk} = \mathsf{Dec}_{K_{enc}}(\epsilon_{ijk})$, with $1 \le k \le m$.
– Re-generate $\delta \in GF(p) : \delta = f_{K_{prf1}}(i)$
– If $\mathtt{t}_{ij} \neq \sum_{(k,\nu_k)\in Q} \nu_k f_{K_{prf3}}(i||j||k||z_{ij1}||\dots||z_{ijm}) + \delta\rho_{ij} \bmod p$, then C declares S_i faulty.

Figure 7: The Challenge **phase of the** RDC-NC **scheme**

GenProofPossession$(Q, \bar{\mathtt{c}}_{ij}, \mathtt{t}_{ij1}, \dots, \mathtt{t}_{ijs})$
// run by a server to compute a proof of possession for the block $\bar{\mathtt{c}}_{ij}$, using the client's query Q, the segments in the block $\mathtt{c}_{ij1}, \dots, \mathtt{c}_{ijs}$, and their associated challenge tags $\mathtt{t}_{ij1}, \dots, \mathtt{t}_{ijs}$.

1. Parse Q as a set of r pairs (k, ν_k) (these pairs correspond to the segments that are being checked, where k is the index of the segment and ν_k is the corresponding query coefficient).
2. Compute $\mathtt{t}_{ij} = \sum_{(k,\nu_k)\in Q} \nu_k \mathtt{t}_{ijk} \bmod p$.
   Compute $\rho_{ij} = \sum_{(k,\nu_k)\in Q} \nu_k \mathtt{c}_{ijk} \bmod p$.
3. Return $(\mathtt{t}_{ij}, \rho_{ij})$.

Figure 8: The GenProofPossession **component of** RDC-NC

B. REPLAY ATTACK AGAINST BASIC NETWORK CODING-BASED SCHEME

Assume the following chain of events in a configuration similar with the one in Fig. 1(c), in which the attacker can corrupt at most one server (out of three) in each epoch.

Epoch 1:

$$\begin{array}{lllll} S_1 & : & \mathtt{x}_{11} = \mathtt{b}_1 & , & \mathtt{x}_{12} = \mathtt{b}_2 + \mathtt{b}_3 \\ & & & & (\textit{i.e.}, S_1 \text{ stores coded blocks } \mathtt{x}_{11}, \mathtt{x}_{12}) \\ S_2 & : & \mathtt{x}_{21} = \mathtt{b}_3 & , & \mathtt{x}_{22} = \mathtt{b}_1 + \mathtt{b}_2 \\ S_3 & : & \mathtt{x}_{31} = \mathtt{b}_1 + \mathtt{b}_3 & , & \mathtt{x}_{32} = \mathtt{b}_2 + \mathtt{b}_3 \end{array}$$

The attacker corrupts S_3, but keeps a copy of $\mathtt{x}_{31}, \mathtt{x}_{32}$ (and their challenge tags).
The client detects corruption at S_3. Thus, it contacts S_1 and S_2 to generate new blocks
$\mathtt{x}'_{31} = 1 \cdot (1 \cdot \mathtt{x}_{11} + 1 \cdot \mathtt{x}_{12}) + 1 \cdot (1 \cdot \mathtt{x}_{21} + 0 \cdot \mathtt{x}_{22}) = \mathtt{b}_1 + \mathtt{b}_2 + 2\mathtt{b}_3$
$\mathtt{x}'_{32} = 1 \cdot (1 \cdot \mathtt{x}_{11} + 0 \cdot \mathtt{x}_{12}) + 1 \cdot (0 \cdot \mathtt{x}_{21} + 1 \cdot \mathtt{x}_{22}) = 2\mathtt{b}_1 + \mathtt{b}_2$
The new blocks $\mathtt{x}'_{31}$ and $\mathtt{x}'_{32}$ are then stored at S_3.
At the end of this epoch, the data recovery condition holds true.

Epoch 2: The attacker corrupts S_1. The client detects corruption at S_1; thus, it contacts $S2$ and S_3 to generate new blocks
$\mathtt{x}'_{11} = 1 \cdot (1 \cdot \mathtt{x}_{21} - 4 \cdot \mathtt{x}_{22}) + 1 \cdot (1 \cdot \mathtt{x}'_{31} + 3 \cdot \mathtt{x}'_{32}) = 3\mathtt{b}_1 + 3\mathtt{b}_3$
$\mathtt{x}'_{12} = 1 \cdot (1 \cdot \mathtt{x}_{21} + 5 \cdot \mathtt{x}_{22}) + 1 \cdot (1 \cdot \mathtt{x}'_{31} - 3 \cdot \mathtt{x}'_{32}) = 3\mathtt{b}_2 + 3\mathtt{b}_3$
The new blocks $\mathtt{x}'_{11}$ and $\mathtt{x}'_{12}$ are then stored at S_1.
The current configuration is now:

$$\begin{array}{lllll} S_1 & : & \mathtt{x}'_{11} = 3\mathtt{b}_1 + 3\mathtt{b}_3 & , & \mathtt{x}'_{12} = 3\mathtt{b}_2 + 3\mathtt{b}_3 \\ S_2 & : & \mathtt{x}_{21} = \mathtt{b}_3 & , & \mathtt{x}_{22} = \mathtt{b}_1 + \mathtt{b}_2 \\ S_3 & : & \mathtt{x}'_{31} = \mathtt{b}_1 + \mathtt{b}_2 + 2\mathtt{b}_3 & , & \mathtt{x}'_{32} = 2\mathtt{b}_1 + \mathtt{b}_2 \end{array}$$

At the end of this epoch, the data recovery condition holds true.

Epoch 3: Attacker corrupts S_3 and replaces blocks $\mathtt{x}'_{31}, \mathtt{x}'_{32}$ with the previously stored blocks $\mathtt{x}_{31}, \mathtt{x}_{32}$. The current configuration is now:

$$\begin{array}{lllll} S_1 & : & \mathtt{x}'_{11} = 3\mathtt{b}_1 + 3\mathtt{b}_3 & , & \mathtt{x}'_{12} = 3\mathtt{b}_2 + 3\mathtt{b}_3 \\ S_2 & : & \mathtt{x}_{21} = \mathtt{b}_3 & , & \mathtt{x}_{22} = \mathtt{b}_1 + \mathtt{b}_2 \\ S_3 & : & \mathtt{x}_{31} = \mathtt{b}_1 + \mathtt{b}_3 & , & \mathtt{x}_{32} = \mathtt{b}_2 + \mathtt{b}_3 \end{array}$$

All the servers successfully pass the integrity check individually. However, in epoch 4, the data recovery condition can be broken and the attacker can cause permanent damage.

Epoch 4:
Attacker corrupts S_2 and the client is not able to create new blocks, because S_1 and S_3 do not collectively store anymore at least 3 linearly independent combinations of the original blocks. Thus, the original file is unrecoverable.