Towards self-repairing replication-based storage systems using untrusted clouds — Source link

Towards Self-Repairing Replication-Based Storage Systems Using Untrusted Clouds

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

ABSTRACT

Distributed storage systems store data redundantly at multiple servers which are geographically spread throughout the world. This basic approach would be sufficient in handling server failure due to natural faults, because when one server fails, data from healthy servers can be used to restore the desired redundancy level. However, in a setting where servers are untrusted and can behave maliciously, data redundancy must be used in tandem with Remote Data Checking (RDC) to ensure that the redundancy level of the storage systems is maintained over time.

All previous RDC schemes for distributed systems impose a heavy burden on the data owner (client) during data maintenance: To repair data at a faulty server, the data owner needs to first download a large amount of data, re-generate the data to be stored at a new server, and then upload this data at a new healthy server. We propose $\mathsf{RDC-SR}$, a novel RDC scheme for replication-based distributed storage systems. $\mathsf{RDC-SR}$ enables Server-side Repair (thus taking advantage of the premium connections available between a CSP's data centers) and places a minimal load on the data owner who only has to act as a *repair coordinator*. The main insight behind $\mathsf{RDC-SR}$ is that the replicas are differentiated based on a controllable amount of masking, which offers $\mathsf{RDC-SR}$ flexibility in handling different adversarial strengths. Also, replica generation must be time consuming in order to avoid certain colluding attacks from malicious servers. Our prototype for $\mathsf{RDC-SR}$ built on Amazon AWS validates the practicality of this new approach.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms

Security, Design, Performance

Keywords

Cloud storage, remote data integrity checking, server-side repair, replicate on the fly, Amazon AWS

1. Introduction

The recent proliferation of cloud services has made it easier than ever to build distributed storage systems based on Cloud Storage Providers (CSPs). Traditionally, a distributed storage system stores data redundantly at multiple servers which are geographically spread throughout the world. In a benign setting where the storage servers always behave in a non-adversarial manner, this basic approach would be sufficient in order to deal with server failure due to natural faults. In this paper however, we consider a setting in which the storage servers are untrusted and may act maliciously. In this setting, Remote Data Checking (RDC) [3, 4, 15, 21] can be used to ensure that the data remains recoverable over time even if the storage servers are untrusted.

When a distributed storage system is used in tandem with remote data checking, we can distinguish several phases throughout the lifetime of the storage system: Setup, $\mathsf{Challenge}$, and Repair. To outsource a file $\mathtt{F}$, the data owner creates multiple replicas of the file during Setup and stores them at multiple storage servers (one replica per server). During the $\mathsf{Challenge}$ phase, the data owner can ask periodically each server to provide a proof that the server's replica has remained intact. If a replica is found corrupt during the $\mathsf{Challenge}$ phase, the data owner can take actions to Repair the corrupted replica using data from the healthy replicas, thus restoring the desired redundancy level in the system. The $\mathsf{Challenge}$ and Repair phases will alternate over the lifetime of the system.

In cloud storage outsourcing, a data owner stores data in a distributed storage system that consists of multiple cloud storage servers. The storage servers may belong to the same CSP (*e.g.*, Amazon has multiple data centers in different locations), or to different CSPs. The ultimate goal of the data owner is that the data will be retrievable at any point of time in the future. Conforming to this notion of storage outsourcing, the data owner would like to outsource *both the storage and the management* of the data. In other words, after the Setup phase, the data owner should only have to store a small, constant, amount of data and should be involved as little as possible in the maintenance of the data. In previous work, the data owner can have minimal involvement in the $\mathsf{Challenge}$ phase when using an RDC scheme that has public verifiability (*i.e.*, the task of verifying that data remains retrievable can be delegated to a third party auditor). However, in all previous work [12, 7], the Repair phase imposes a significant burden on the data owner, who needs to expend a significant amount of computation and communication. For example, to repair data at a failed server, the data owner needs to first download an amount of data equal to the file size, re-generate the data to be stored at a new server, and then upload this data at a new healthy server ([12, 7]). Archival storage deals with large amounts of data (Terabytes or Petabytes) and thus maintaining the health of the data imposes a heavy burden on the data owner.

In this work, we ask the question: Is it possible to design an RDC scheme which can repair corrupted data with the least data owner intervention? We answer this question in the positive by exploring a model which minimizes the data owner's involvement in the Repair phase, thus fully realizing the vision of outsourcing both the storage and management of data. During Repair, the data owner simply acts as a *repair coordinator*, which allows the data owner to manage data using a lightweight device. This is in contrast with previous work, which imposes a heavy burden on the data owner during Repair. The main challenge is how to ensure that the untrusted servers manage the data properly over time (*i.e.*, take necessary actions to maintain the desired level of redundancy when some of the replicas have failed).

Main objective: Informally, our main objective is to design an RDC scheme for a replication-based distributed storage system which has the following properties:

– the system stores t replicas of the data owner's original file
– the system imposes a small load on the verifier during the Challenge phase.
– the system imposes a small management load on the data owner (by minimizing the involvement of the data owner during the Repair phase).

The first two properties alone can be achieved based on techniques proposed in previous work ([12] provides multiple replica guarantees, whereas RDC based on spot-checking [3, 15, 21] supports a lightweight verification mechanism in the Challenge phase). The challenge is to achieve the third property without giving up any of the first two properties. We meet these objectives by proposing a new model and by re-designing the three phases of a traditional RDC protocol.

1.1 Solution overview

Two insights motivate the design of our solution:

Insight 1. Replica differentiation: The storage servers should be required to store t different replicas. Otherwise, if all replicas are identical, an economically motivated set of colluding servers could attempt to save storage by simply storing only one replica and redirect all client's challenges to the one server storing the replica.

Previous work [6, 17] proposed to store identical replicas at storage servers which are in different locations. To check that each server stores a replica, they require servers to respond fast, thus relying on the network delay and bandwidth properties. While storing identical replicas has the advantage of simplicity, in Sec. 2.1 we show that this approach has major limitations. Moreover, we show that for real-world CSPs, one of the assumptions made by [6] does not hold.

Insight 2. Server-side repair: We can minimize the load on the data owner during the Repair phase by relying on the servers to collaborate in order to generate a new replica whenever a replica has failed. This is advantageous because of two reasons:

(a) the servers are usually connected through premium network connections (high bandwidth), as opposed to the data owner's connection which may have limited download/upload bandwidth. Our experiments in Table 2 (Appendix A) show that Amazon AWS has premium Internet connection of up to tens of MB/s between its data centers.
(b) the computational burden during the Repair phase is shifted to the servers, allowing data owners to remain lightweight.

Previous RDC schemes for replication-based distributed storage systems ([12]) do not give the storage servers access to the original data owner's file. Each replica is a masked/encrypted version of the original file. As a result, the Repair phase imposes a high burden on the data owner: The communication and computation cost to create a new replica is linear with the size of the replica because the data owner needs to download a replica, unmask/decrypt it, create a new replica and upload the new replica. If the servers do not have access to the original file, this intense level of data owner involvement during Repair is unavoidable.

In this paper, we propose to use a different paradigm, in which the data owner gives the servers both access to the original file and the means to generate new replicas. This will allow the servers to generate a new replica by collaborating between themselves during Repair.

A Basic Approach and Its Limitations. A straightforward approach would be for the data owner to create *different* replicas by using masking/encryption of the original file. The data owner would reveal to the servers the key material used to create the masked/encrypted replicas. During Repair, the servers themselves could recover the original file from a healthy replica and restore the corrupted replica, reducing the burden on the data owner.

This basic approach is vulnerable to a potential attack, the *replicate on the fly (ROTF) attack*: During Repair, a malicious set of servers could claim they generate a new replica whenever an existing replica has failed, but in reality they do not create the replica (using this strategy, an economically motivated set of servers tries to use less storage than their contractual obligation). When the client checks the newly generated replica during the Challenge phase, the set of malicious servers can collaborate to generate the replica *on the fly* and pass the verification successfully (this replica is then immediately deleted after passing the challenge in order to save storage). This will hurt the reliability of the storage system, because in time the system will end up storing much fewer than t replicas, unbeknownst to the client.

Overcoming the ROTF attack. The new paradigm we introduce in this paper, which allows the servers to generate a new replica by collaborating between themselves during Repair, has the important advantage of minimizing the load on the data owner during data maintenance. However, this comes at the cost of allowing a new attack avenue for servers, the ROTF attack.

To overcome the ROTF attack, we *make replica creation to be time consuming.* In this way, malicious servers cannot generate replicas on the fly during Challenge without being detected.

Contributions. In this paper, we propose RDC − SR, a novel RDC scheme for replication-based distributed storage systems, which enables Server-side Repair. Compared to all the previous distributed RDC schemes, which impose a high load on the data owner in the Repair phase, our RDC − SR scheme imposes a small load on both the verifier in the Challenge phase and the data owner in the Repair phase. To the best of our knowledge, we are the first to propose the server-side repair strategy and an RDC scheme that implements it in the context of a real-world CSP. Specifically, our paper makes the following contributions:

- We point out limitations of a previous network delay-based model for establishing data geolocation and revise this model to suit our approach. Based on experiments on the Amazon cloud platform, we show that one of the assumptions made in the network delay-based model does not hold in practice. We further show that an RDC scheme built on such a model can only provide a very low data possession assurance.

- We revise this model to include replica differentiation and time-consuming replica generation, in order to limit the ability of economically-motivated adversaries to cheat. In this new model, in which servers are allowed to generate new replicas, the burden during the Repair phase is shifted to the server side, allowing lightweight clients to perform data maintenance. We observe that this new paradigm enables a new attack, the *replicate on the fly (ROTF)* attack, which requires us to consider a new adversarial model, the α-cheating adversary that seeks to cheat by only storing an α fraction of its contractual storage obligations.

- All previous distributed RDC schemes place a heavy burden on the client during repair. We propose RDC − SR, a novel RDC scheme for replication-based distributed storage systems. RDC − SR enables Server-side Repair (thus taking advantage of the premium connections available between a CSP's data centers) and places a minimal load on the client who only has to act as a *repair coordinator*. To integrate our new model into RDC − SR, we devise a novel technique by which replicas are differentiated based on a controllable amount of masking; this offers RDC − SR flexibility in handling different adversarial strengths. We prove that RDC − SR can mitigate the ROTF attack.

- We provide guidelines on how to choose the parameters for RDC − SR in a practical setting and build a prototype for RDC − SR on Amazon AWS. The experimental results show that: (a) RDC − SR imposes only a small load on the verifier in the Challenge phase and a small management load on the data owner in the Repair phase; (b) RDC − SR can easily differentiate benign and adversarial CSP behavior when relying on a time threshold, as 95% of the benign cases are under the threshold, while 100% of the adversarial cases are over the threshold.

2. Models for Checking Replica Storage

In this section, we first review a previously proposed theoretical framework that relies purely on network delay to establish the geolocation of files at cloud providers, and point out several limitations of this model when used with a basic RDC protocol on the Amazon cloud service provider. The main limitation is that one of its assumptions does not hold in a practical setting, and thus a protocol that relies only on the network delay to detect server misbehavior can only offer a very low data possession guarantee. We then augment this model to include time-consuming replica generation in order to make RDC usable for geolocation of files in the context of a real-world cloud storage provider such as Amazon.

2.1 A Network Delay-based Model and Its Limitations

Benson, Dowsley and Shacham proposed a theoretical model for verifying that copies of a client's data are stored at

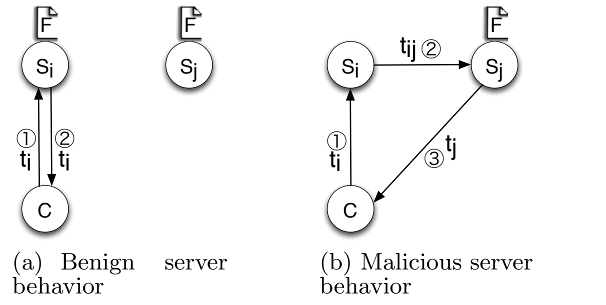


Figure 1: Auditing protocol: Client C checks if server s_i has a file copy F.

different geographic locations [6] (we refer to it as the "BDS model"). This model allows to derive a condition which can be used to detect if a server at some location does not have a copy of the data. The idea behind the condition is that an auditor which challenges a storage server must receive an answer within a certain time, otherwise the server is considered malicious. The time is chosen such that a server that does not have the challenged data cannot provide an answer by using data from a server at a different geolocation.

The BDS model [6]. The customer (client) makes a contract with the CSP to store one copy of the client's file in each of the CSP's k data centers. For simplicity, if we assume that $k = 2$, then a file copy should be stored at s_i and another file copy at s_j. The goal is to build an audit protocol that tests if the cloud provider is really storing one copy of the file in each of the two data centers s_i and s_j. Several assumptions need to be made:

(Assumption 1) The locations of all data centers of the cloud provider are known.

(Assumption 2) The cloud provider does not have any exclusive Internet connection between the data centers.

(Assumption 3) For each datacenter s, it is possible to have access to a machine that is located very close to s (*i.e.*, very small network latency), such as in the same data center.

Consider the case when the client wants to check if s_i is storing a copy of the file. As shown in Figure 1(b), s_i and s_j may be colluding malicious servers who only store one copy of the file at s_j; when s_i is challenged by the client to prove data possession, it redirects the challenge to s_j, who answers directly to the client. To prevent such an attack, the client imposes a certain time limit for receiving the answer.

Let T_i be the upper bound on the execution time of some auditing protocol by a datacenter s_i, t_i be the network delay between the client and s_i, and t_{ij} be the network delay between data centers s_i and s_j. For a network delay time t, we use the notation $max(t)$ to denote the upper bound on t and $min(t)$ to denote the lower bound on t.

If the data center s_i is honest, the client accepts the audit protocol execution as valid if the elapsed time for receiving the answer is $T_i + 2 * max(t_i)$, because that is the time needed to receive the answer in the worst case scenario. On the other hand, if the answer is received after $min(t_i) + min(t_{ij}) + min(t_j)$, the client should consider the audit protocol execution invalid, since s_i may be dishonest and may be using data from another data center. Thus

$$T_i + 2 * max(t_i) \leq min(t_i) + min(t_{ij}) + min(t_j), \text{ or}$$

$$T_i \leq min(t_i) + min(t_{ij}) + min(t_j) - 2 * max(t_i) \quad (1)$$

Limitations of the basic PoR protocol based on BDS model. Based on the condition derived from the BDS model, [6] proposed a basic Proof of Retrievability (PoR) protocol which seeks to ensure that a set of storage servers not only store n copies of the client's data, but also that these copies are stored at specific geographic locations known to the client. In the PoR protocol, the client stores identical copies of a file at multiple storage servers, and for each copy, it also stores authentication tags (one tag for each file block). To check that a server has a copy of the file, the auditor asks the server to provide several randomly chosen file blocks and their corresponding MAC tags. If the auditor receives the answer within a certain time, the auditor checks if the MAC tags are valid tags for the file blocks. In this protocol, the auditor challenges as many random blocks as it is possible for s_i to access in time T_i.

Based on Assumption 3 in the BDS model, the auditor can be located very close to s_i, which means that $min(t_i)$ and $max(t_i)$ will be small compared to t_{ij} and t_j. Thus, the value of T_i will be mainly determined by $min(t_{ij})$ and $min(t_j)$, which is determined by the quality (bandwidth) of the Internet connection between s_i and s_j and by the distance between s_i and s_j. Low bandwidth Internet connection and large distance between s_i and s_j will result in larger values of $min(t_{ij})$ and $min(t_j)$, thus resulting in a larger T_i. A larger T_i means the auditor can challenge more blocks while still being able to differentiate a benign server from a malicious server (the auditor should be able to challenge a large enough number of randomly chosen blocks in order to gain a reasonable confidence that the entire file is stored by the server).

To ensure that T_i is large enough (and thus the protocol has practical value), the BDS model relies explicitly on the assumption that there is no exclusive Internet connection between data centers (Assumption 2). The BDS model also relies on the implicit assumption that the data centers should be far away from each other. However, our measurements with the Amazon CSP show that these assumptions do not hold (see Tables 2 and 3 in Appendix A). In general, the network delay is the sum between propagation delay (the time it takes the signal to travel from sender to the receiver) and the transmission delay (the time required to push all the data bits into the wire). From Table 2, we can see that the download bandwidth between different S3 data centers varies between 11-36 MB/s, which is significantly higher than the bandwidth between a point outside the data centers and the data centers (less than 1 MB/s between our institution and different S3 data centers). We also notice that inside a data center the download bandwidth is very high (between 32-52 MB/s) and the propagation delay is very small (between 0.2-0.7 milliseconds). Finally, we notice from Table 3 that the propagation delay between certain S3 data centers is quite small (*e.g.*, 11 milliseconds between N. California and Oregon).

Using the numbers in Tables 2 and 3, with s_i and s_j being the Virginia and the N. California data centers respectively, and assuming that the auditor is located within s_i and challenges k 4KB random file blocks from s_i, Equation (1) for the basic PoR protocol becomes $x \cdot k \leq 80 + 0.3k$, where x is the time to access one random file block. According to our experiments on Amazon S3, $x \approx 30$ milliseconds (refer to Appendix C), thus $k \leq 2.66$. This means the basic PoR protocol applied in the setting of the Amazon CSP allows the auditor to challenge at most two random file blocks in each protocol execution. This provides a very low data possession assurance (comparatively, to achieve 99% confidence that misbehavior will be detected when the server corrupts 1% of the file, the auditor should challenge 460 randomly chosen file blocks [3]).

2.2 A New Model to Enable Server-side Repair

The main problem with the basic PoR protocol based on the BDS model (cf. Sec. 2.1) is that all the file copies are identical and the auditor relies solely on the network delay to detect malicious server behavior. As a result, the protocol must assume that there is no exclusive Internet connection between the data centers (Assumption 2 in the BDS model). Having established in Section 2.1 that this assumption does not hold in a practical setting, we augment the BDS model to make it usable in a practical setting. Namely, we require that the file replicas stored at each server must be different and personalized for each server. Upon being challenged, each server must produce an answer that is dependent on its own replica. As a result, a server cannot answer a challenge by using another server's replica. An economically-motivated server who does not possess its assigned replica may try to cheat by using another server's replica. But to do this, the cheating server must first generate its own replica in order to successfully answer a challenge. As a result, our model does not rely purely on network delay to differentiate benign behavior from malicious behavior, but also on the time it takes to generate a file replica. This allows us to eliminate Assumption 2 from the BDS model, because we require that *replica generation be time consuming*.

We propose a model in which the client creates t different file replicas and stores them at t data centers owned by the same CSP (one replica at each data center). To illustrate the model for $t = 2$, the data owner generates file replicas $\mathtt{F}_i$ and $\mathtt{F}_j$; server s_i stores $\mathtt{F}_i$ and s_j stores $\mathtt{F}_j$. Even when replicas are different, malicious servers may execute the ROTF attack, in which a server that does not possess its assigned replica may try to cheat by using replicas from other servers to generate its assigned replica on the fly during the Challenge phase. Using the same notation as in the BDS model in Section 2.1, an audit protocol execution should be considered invalid if the answer is received after $min(t_i) + min(t_{ij}) + min(t_j) + min(t_R)$, where t_R denotes the time required to generate replica $\mathtt{F}_i$ (more precisely, the time required to generate the portion of $\mathtt{F}_i$ that is necessary to construct a correct answer). Thus, the condition used to differentiate benign from malicious behavior becomes:

$$T_i + 2 * max(t_i) \leq min(t_i) + min(t_{ij}) + min(t_j) + min(t_R) \quad (2)$$

We only need to make the following two assumptions (note that we do not assume there is no exclusive Internet connection between the data centers like in the BDS model):

(Assumption 1) The locations of all datacenters of the cloud provider are known.

(Assumption 2) For each datacenter s, it is possible to have access to a machine that is located very close to s (*i.e.*, very small network latency), such as in the same data center.

3. System and Adversarial Model

3.1 System Model

The client wants to outsource the storage of a file F. To ensure high reliability and fault tolerance of the data, the client creates t *distinct* replicas and outsources them to t data centers (storage servers) owned by a CSP (one replica at each data center). To ensure that the t replicas remain healthy over time, the client challenges each of the t servers periodically. Upon finding a corrupted replica, the client acts as a *repair coordinator* who oversees the repair of the corrupted replica (the CSP, who has premium network connection between its data centers, uses the healthy replicas to repair the corrupted replica; the client should have minimal involvement in the repair process).

We note that, given an individual file replica, say $\mathtt{F}_i$, the CSP can generate any another replica, say $\mathtt{F}_j$, in two steps: first recover the original file F from $\mathtt{F}_i$, and then generate $\mathtt{F}_j$.

3.2 Adversarial model

We assume that the CSP is rational and economically motivated. The CSP will try to cheat only if cheating cannot be detected and if it achieves some economic benefit, such as using less storage than required by contract. An economically motivated adversary captures many practical settings in which malicious servers will not cheat and risk their reputation, unless they can achieve a clear financial gain. We also note that when the adversary is fully malicious, *i.e.*, it tries to corrupt the client's data without regard to its own resource consumption, there is no solution to the problem of building a reliable system with t replicas [7, 12].

The ROTF attack. We are particularly concerned with the following *replicate on the fly (ROTF) attack*: During Repair, a set of colluding servers could claim they generate a new replica whenever an existing replica has failed, but in reality they do not create and store the replica. When the client checks the newly generated replica during the Challenge phase, the set of malicious servers can collaborate to generate the replica on the fly and pass the verification successfully. Immediately after the check, the servers delete the newly generated replica, only to re-generate it on the fly when the client initiates the next check. This will hurt the reliability of the storage system, because in time the system will end up storing much fewer than t replicas, unbeknownst to the client.

To illustrate the ROTF attack, consider the setting in Figure 1(b), where s_i and s_j should to store replicas $\mathtt{F}_i$ and $\mathtt{F}_j$, respectively, but only s_j stores $\mathtt{F}_j$. When s_i is being challenged to prove possession of $\mathtt{F}_i$, s_i can retrieve $\mathtt{F}_j$ from s_j, and generate $\mathtt{F}_i$ on the fly in order to pass the challenge. Or, it can forward the challenge to s_j, who uses $\mathtt{F}_j$ to generate on the fly $\mathtt{F}_i$ and then uses $\mathtt{F}_i$ to construct a valid response to the challenge. Immediately after the challenge, $\mathtt{F}_i$ is deleted.

The α-cheating adversary. A CSP is obligated by contract to store t file replicas, which requires a total of $t|\mathtt{F}|$ storage. However, a dishonest CSP may try to use less than $t|\mathtt{F}|$ storage space and hope that this will go undetected (*e.g.*, executes the ROTF attack). We use the following definition to denote a CSP that is using only an α fraction of its contractual storage obligation:

Definition 3.1. *An* α-cheating adversary *is an economically-motivated adversary that can successfully pass a challenge by only using* $\alpha t|F|$ *storage (where* $\frac{1}{t} \leq \alpha \leq 1$*).*

Note that if $\alpha < \frac{1}{t}$, then the CSP stores less than $|\mathtt{F}|$, which means that any single-replica RDC scheme [3, 15] would be enough to detect the CSP's dishonest behavior. Thus, we do not consider the case when $\alpha < \frac{1}{t}$.

Adversarial Strategies. Replica generation in our model is time consuming. A dishonest CSP trying to cheat by storing less replicas and executing the ROTF attack is always better off by keeping a copy of the original file F. While this strategy requires some additional storage, it increases considerably the CSP's chances to cheat undetectably because the CSP can generate any individual replica from F in one step. Otherwise, cheating would require a two-step process: To generate a particular replica that is being challenged and which is not in its possession, say $\mathtt{F}_i$, the CSP would need to first recover F from an existing replica, say $\mathtt{F}_j$, and then generate $\mathtt{F}_i$ from F. Since replica generation is a time consuming operation (and similarly recovering F from one of its replicas is also time-consuming), this two-step process would considerably increase the client's chances of detecting the CSP's dishonest behavior. Thus, we assume a dishonest CSP always stores a copy of the original file F.

Also, recall that most RDC schemes ensure efficiency by using spot checking [3, 15, 21]: The client challenges the server to prove possession of a randomly chosen subset of c blocks out of all the n file blocks. This can provide a high likelihood that the server is storing the entire file.

An α-cheating adversary can adopt several strategies to distribute its $\alpha t|\mathtt{F}|$ storage among the t servers, which will influence its ability to remain undetected. A *basic strategy* is when the adversary chooses to store on one of the servers the original file F, and to store on each of $\lfloor \alpha t \rfloor - 1$ servers a particular different replica. Thus, no data is stored on the remaining $t - \lfloor \alpha t \rfloor$ servers. In this case, when one of the $t - \lfloor \alpha t \rfloor$ servers is challenged, it can always generate the c challenged blocks on the fly and then construct the answer to the challenge. Let σ be the time required to generate the c challenged blocks for one replica.

It turns out that the *best data distribution strategy for cheating* is when the adversary stores in each of the t servers an α fraction of the blocks from the corresponding replica for that server. Thus, the adversary will still only use $\alpha t|\mathtt{F}|$ storage space in total[1]. When any of the t servers is challenged, this server will already possess, on average, an α fraction of the c blocks that are being challenged. Thus, on average, it only needs to generate on the fly a $(1-\alpha)$ fraction of the c challenged blocks, which it can do in time $(1-\alpha)\sigma$. Since, $(1-\alpha)\sigma < \sigma$, this strategy is always better than the previously presented basic strategy.

4. An RDC Scheme with Server-side Repair

In this section, we propose $\mathsf{RDC-SR}$, the first replication-based Remote Data Checking scheme that supports Server-side Repair.

The original file F has n blocks, $\mathtt{F} = \{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$, and each contains s symbols in $GF(p)$, where p is a large prime (at least 80 bits). We use j to denote the index of a block within a file / replica (*i.e.*, $j \in \{1 \ldots n\}$), and k to denote the index of a symbol in a block (*i.e.*, $k \in \{1 \ldots s\}$). Let κ be

[1]Recall that we have assumed that the adversary always stores one original file copy F, thus the total storage is $(\alpha t + 1)|\mathtt{F}|$; when t is large, this can be approximated by $\alpha t|\mathtt{F}|$.

a security parameter. We make use of two pseudo-random functions (PRFs) h and γ with the following parameters:
$- h : \{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\log p}$

$- \gamma : \{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\log p}$

RDC − SR overview. Like any RDC system for a multiple-server setting [12, 7, 24, 11], RDC − SR consists of three phases: Setup, Challenge and Repair. During the Setup phase, the client first preprocesses the original file and generates t distinct replicas. We use i to denote the index of the replica (*i.e.*, $i \in \{1 \dots t\}$). To differentiate the replicas, we adopt a masking strategy similar as in [12], in which every symbol of the original file is masked individually by adding a random value modulo p. We introduce a new parameter η, which denotes the number of masking operations imposed on each symbol when generating a distinct replica. η can help control the computational load caused by the masking, *e.g.*, we can choose a larger η if we try to make the masking more expensive for a block. This has the advantage that we can adjust the load for masking to defend against different adversarial strenghts (see Sec. 3.2). The client then generates verification tags for every replica, one tag per file block. Each verification tag is computed similarly as in [21], namely as a message authentication code by combining universal hashing with a PRF [16, 20, 22, 27]. After having generated t distinct replicas and the corresponding verification tags, the client sends those replicas to t different data centers of the CSP (one replica per server), and the set of all verification tags to each data center. The client also makes public the key used for generating the distinct replicas, so that the servers can use it during Repair to generate new replicas on their own.

During the Challenge phase the client acting as the verifier uses spot checking to check the replica at each server s_i, in which it randomly samples a small subset of blocks from a replica and checks their validity based on the the server s_i's response. Such a technique can detect replica corruption with high probability [3], and has the advantage of only imposing a small overhead on both the client and the server. We use a threshold τ for our new model (see Sec.2.2): If the response from a server is not received within time τ, then that replica will be considered corrupted.

The Repair phase is activated when the verifier has detected a corrupted replica during Challenge. The client acts as the repair coordinator, *i.e.*, it coordinates the CSP's servers to repair the corruption. We take advantage of the fact that a CSP usually has premium bandwidth between its data centers (refer to Table 2) and permit the servers to collaborate among themselves to restore the corrupted replica (the key for generating distinct replicas is known to the CSP). Thus, the system only imposes a small management load on the client (data owner).

A detailed description of RDC − SR is provided in Figures 2 and 3, together with the following explanation of the three phases.

The Setup phase. The client first generates keys K_1 and K_2. K_1 will be used to compute the verification tags and K_2 will be used in generating distinct replicas. It then picks s random numbers, η, and threshold τ (refer to Sec. 5 – Parameterization and Guidelines on how to exactly determine η and τ). The client then calls GenReplicaAndMetadata t times in order to generate t distinct replicas and the corresponding verification tags. Each replica will be sent to a server located in a different data center of the CSP. The entire set of verification tags will be sent to each server. The client may then delete the original file and only keep a small amount of key material.

In GenReplicaAndMetadata, the client masks the original file at the symbol level, applying η masking operations to each symbol. Each masking operation consists of adding a pseudo-random value to the symbol; this pseudo-random value is the output of a PRF applied over the concatenation of the replica index, the block index, the symbol index, and an integer l ($l \in \{1 \dots \eta\}$).

The Challenge phase. For this phase, we integrate spot checking [3, 15, 21] with our new model introduced in Sec. 2.2. The client (verifier) sends a challenge request to

We construct RDC − SR in three phases, Setup, Challenge, and Repair. All arithmetic operations are in $GF(p)$, unless noted otherwise explicitly.

Setup: The client runs $(K_1, K_2) \leftarrow \mathsf{KeyGen}(1^{\kappa})$, and picks s random numbers $\delta_1, \dots, \delta_s \overset{R}{\leftarrow} GF(p)$. The client also chooses α and determines the values η and τ, and then executes:
For $1 \leq i \leq t$:

1. Run $(\mathtt{t}_{i1}, \dots, \mathtt{t}_{in}, \mathtt{F}_i) \leftarrow \mathsf{GenReplicaAndMetadata}(K_1, K_2, \mathtt{F}, i, \delta_1, \dots, \delta_s, \eta)$
2. Send $\mathtt{F}_i$ to server S_i for storage (each S_i is located in a different data center of the CSP) and send the verification tags $\mathtt{t}_{i1}, \dots, \mathtt{t}_{in}$ to each server .

The client may now delete the file $\mathtt{F}$ and stores only a small, constant, amount of data: K_1, $\delta_1, \dots, \delta_s$, η, and τ. K_2 is made public.

Challenge: Client C uses spot checking to check possession of each replica $\mathtt{F}_i$ stored at server S_i. In this process, each server uses its stored replica and the corresponding verification tags to prove data possession. Suppose C challenges server S_i. Let query Q be the c-element set $\{(j, v_j)\}$, in which j denotes the index of the block to be challenged, and v_j is a randomly chosen value from $GF(p)$.

1. C generates Q and sends Q to server S_i
2. S_i runs $(\rho_1, \dots, \rho_s, \mathtt{t}) \leftarrow \mathsf{GenProof}(Q, \mathtt{F}_i, \mathtt{t}_{i1}, \dots, \mathtt{t}_{in})$
3. S_i sends to C the proof of possession $(\rho_1, \dots, \rho_s, \mathtt{t})$
4. C checks whether the response time is larger than τ. If yes, C declares S_i as faulty. Otherwise, C checks the validity of the proof $(\rho_1, \dots, \rho_s, \mathtt{t})$ by running $\mathsf{CheckProof}(K_1, \delta_1, \dots, \delta_s, Q, \rho_1, \dots, \rho_s, \mathtt{t}, i)$

Repair: Assume that in the Challenge phase C has identified a faulty server whose index is y (*i.e.*, the corresponding replica has been corrupted). C acts as the repair coordinator. It communicates with the CSP, asks for a new server from the same data center to replace the corrupted server, and coordinates from where the new server can retrieve a healthy replica to restore the corrupted replica. Suppose S_i is selected to provide the healthy replica. The new server will reuse the index of the faulty server, namely, y.

1. Server S_y retrieves the replica $F_i = \{\mathtt{m}_{i1}, \dots, \mathtt{m}_{in}\}$ and all the verification tags from server S_i
2. Server S_y generates its own replica:
   For $1 \leq j \leq n$:
   - For $1 \leq k \leq s$: $\mathtt{m}_{yjk} = \mathtt{m}_{ijk} - \sum_{l=1}^{\eta} \gamma_{K_2}(i||j||k||l) + \sum_{l=1}^{\eta} \gamma_{K_2}(y||j||k||l)$

Figure 2: RDC − SR: a replication-based RDC system with Server-side Repair

```
KeyGen($1^\kappa$): Randomly choose two keys: $K_1, K_2 \xleftarrow{R} \{0,1\}^\kappa$.
Return $(K_1, K_2)$
GenReplicaAndMetadata($K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s, \eta$):
1. Parse $\mathtt{F}$ as $\{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$
2. Generate the $i$-th replica:
   For $1 \le j \le n$:
     • Mask block $\mathtt{b}_j$ at the symbol level and get $\mathtt{m}_{ij}$:
       For $1 \le k \le s$: $\mathtt{m}_{ijk} = \mathtt{b}_{jk} + \sum_{l=1}^{\eta} \gamma_{K_2}(i||j||k||l)$
3. Compute verification tags:
   For $1 \le j \le n$: $\mathtt{t}_{ij} = h_{K_1}(i||j) + \sum_{k=1}^{s} \delta_k \mathtt{m}_{ijk}$
4. Return $(\mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}, \mathtt{F}_i = \{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\})$
GenProof($Q, \mathtt{F}_i, \mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}$):
1. Parse $Q$ as a set of $c$ pairs $(j, v_j)$. Parse $\mathtt{F}_i$ as
   $\{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\}$.
2. Compute $\rho$ and $\mathtt{t}$:
     • For $1 \le k \le s$: $\rho_k = \sum_{(j,v_j) \in Q} v_j \mathtt{m}_{ijk} \ mod\ p$
     • $\mathtt{t} = \sum_{(j,v_j) \in Q} v_j \mathtt{t}_{ij} \ mod\ p$
3. Return $(\rho_1, \ldots, \rho_s, \mathtt{t})$
CheckProof($K_1, \delta_1, \ldots, \delta_s, Q, \rho_1, \ldots, \rho_s, \mathtt{t}, i$):
1. Parse $Q$ as a set of $c$ pairs $(j, v_j)$
2. If $\mathtt{t} = \sum_{(j,v_j) \in Q} v_j h_{K_1}(i||j) + \sum_{k=1}^{s} \delta_k \rho_k \bmod p$, re-
   turn "success". Otherwise return "failure".
```

Figure 3: Components of $\mathsf{RDC-SR}$

each of the t servers. For each challenge, the client selects c random replica blocks for checking. The challenged server parses the request, calls GenProof to generate the proof, and sends back the proof. If the client does not receive the proof within time τ, it marks that particular server as faulty and its replica as corrupt. Otherwise, the client checks the validity of the proof by calling CheckProof.

The Repair phase. During the Repair phase, the client acts as the repair coordinator; our approach here is novel compared to previous work, in which the client itself repairs the data by downloading the entire file to regenerate a corrupt replica [12, 7, 11]. The client contacts the CSP, reports the corruption, and coordinates the CSP's servers to repair the corruption. The server which is found faulty in the Challenge phase should be replaced by a new server from the same data center. The new server contacts one of the healthy servers, retrieves a replica, un-masks it to restore the original file, and masks the original file to regenerate the corrupted replica. The new server directly retrieves the entire set of verification tags from this healthy server (recall that the entire set of verification tags is stored at every server). Note that the size of the set of all verification tags is always small compared to the data.

5. Guidelines for using $\mathsf{RDC-SR}$

In order to setup the system, the data owner must initially decide the type of adversary it wants to protect the data against. Concretely, by picking a value for α, the data owner seeks to protect its data against a CSP that is modeled as an α-cheating adversary. For example, by picking a small α, the data owner achieves protection against a CSP that will try to cheat by corrupting a large amount of the data. This type of corruption is easier to detect and, as a result, the data owner can afford to use a smaller masking factor. On the other hand, by picking a large α, the data owner seeks protection against a more stealthy CSP that only corrupts a small fraction of the data. As a result, the data owner needs to use a larger masking factor.

Once the data owner fixes α, it can derive the two parameters: η (the masking factor) and τ (the time threshold used to validate the audit protocol).

Estimating η. From Sec. 3, we have $T_i \le min(t_i) + min(t_{ij}) + min(t_j) + min(t_R) - 2 * max(t_i)$, which can be further simplified as $T_i \le t_{ij} + t_j - t_i + t_R$ (Let x be the time each of the c challenged file blocks contributes to the generation of the proof by the server. Since T_i is the upper bound on the execution time of the auditing protocol (as defined in Sec. 2.1), we have $c \cdot x \le T_i$. Based on the triangle inequality, we always have $t_{ij} + t_j - t_i \ge 0$. To have a coarse evaluation of η, we neglect $t_{ij} + t_j - t_i$, which is always small compared to t_R (milliseconds compared to seconds, as shown in Table 4 of Appendix B which contains some typical values based on our experiments for Amazon S3). Thus, we get $c \cdot x \le t_R$.

Let t_{prf} denote the time required to compute one PRF (specifically, one computation of the function γ used to mask a symbol in $\mathsf{RDC-SR}$). Then, for a challenge that checks c blocks, assuming that the adversary adopts the best attack strategy (see Sec. 3.2), we have $t_R = (1-\alpha) \cdot c \cdot s \cdot \eta \cdot t_{prf}$. We thus get $c \cdot x \le t_R = (1-\alpha) \cdot c \cdot s \cdot \eta \cdot t_{prf}$, which means that $\eta \ge \frac{x}{(1-\alpha) \cdot s \cdot t_{prf}}$ (recall that s is the number of symbols in a file block). The client should choose η as the smallest integer which satisfies this condition.

Estimating τ. The time threshold τ can be computed as $c \cdot x + 2 \cdot t_i$. As defined earlier in this section, x denotes the time each of the c challenged file blocks contributes to the generation of the proof by the server, which should include the time for accessing one block and computing the proof for one block. t_i denotes the network delay between the challenged server and the client.

It turns out it is not trivial to estimate x for the Amazon CSP. In our experiments, the value x exhibits some variation due to the fact that sampling a random block in Amazon S3 can be very large in some rare cases (in those cases it will be difficult to differentiate between benign and malicious CSP behavior). However, based on our experiments we observed that, out of 240 protocol executions, 95% of the values for x are within the range [0.025 sec, 0.034 sec] for the AWS Oregon region. Thus, the data owner should use the top value in this range (0.034 sec) to estimate x in the formula for τ if the data is stored in the Oregon S3 region. We propose three ways in which the data owner can acquire x: First of all, data owners can estimate x themselves by measuring it directly in the target data centers; Secondly, the CSP could determine such a range and publish it; Thirdly, it can be estimated by a trusted third party. Note that if x is estimated by data owners or trusted third parties, the CSP should not be able to differentiate the events of "estimating x" and "regular data access", thus it cannot affect the effectiveness of verification by artificially manipulating the value of x.

6. Security Analysis for $\mathsf{RDC-SR}$

Our $\mathsf{RDC-SR}$ scheme is an RDC scheme and it can be easily shown that, in the context of each individual server that holds a replica, $\mathsf{RDC-SR}$ provides the data owner with a guarantee of data possession of that replica by using an efficient spot checking mechanism [3, 15]. Note that confidentiality of the data from the CSP is an orthogonal problem to RDC (although our $\mathsf{RDC-SR}$ scheme could easily achieve confidentiality by encrypting the original file and then storing masked replicas of the encrypted file).

As opposed to previous work on RDC, the paradigm we introduce in this paper allows the servers themselves to generate new replicas for repair purposes. This opens the door to a new attack, the *replicate on the fly (ROTF) attack*, in which the economically-motivated servers claim to store t replicas, but in reality they store less than $t|\mathtt{F}|$ data and generate the missing data on the fly upon being challenged by the client. The following theorem shows that $\mathsf{RDC-SR}$ can mitigate the ROTF attack executed by an α-cheating adversary (defined in Sec. 3.2):

Theorem 6.1. *In* $\mathsf{RDC-SR}$*, an* α*-cheating adversary can successfully execute the ROTF attack without being detected with a probability of at most* $\alpha^c c(1-\alpha)$*, where* c *is the number of file blocks checked by the client in a challenge.*

For fixed values of α, we can always choose c such that the probability that a server is cheating successfully without being detected becomes negligibly small. For example, if a server is storing only 90% of the data (*i.e.*, $\alpha = 0.9$), challenging $c = 400$ random blocks, ensures that the upper bound on the probability of server cheating is $1.99 * 10^{-17}$.

Proof. Per Definition 3.1, an α-cheating adversary is an economically-motivated adversary that only uses $\alpha t|\mathtt{F}|$ storage (where $1/t \leq \alpha \leq 1$). We have established in Sec. 3.2 that the best data distribution strategy for cheating is when each malicious server stores only an α fraction of the blocks from the replica it is supposed to store. Thus each malicious server is missing an $(1-\alpha)$ fraction of the file blocks.

As described in Sec. 5, the time threshold τ in $\mathsf{RDC-SR}$ is computed based on the assumption that every time the client randomly checks c blocks from a file stored in one of the t servers, at least $(1-\alpha)c$ blocks are from the missing $(1-\alpha)$ fraction of the file, and thus the server has to compute $(1-\alpha)c$ blocks on the fly. However, if the number of checked blocks from the $(1-\alpha)$ missing fraction is less than $(1-\alpha)c$, then the cheating server will be able to successfully pass the check because it has to generate less than $(1-\alpha)c$ blocks on the fly and can provide a reply in a time less than τ.

When a server is missing an $(1-\alpha)$ fraction of the file blocks and the client randomly challenges c blocks, let P be the probability that less than $(1-\alpha)c$ blocks will be challenged among the missing file blocks. This is the probability that the cheating server is able to cheat successfully without being detected. We evaluate P next.

Evaluating P is equivalent to evaluating the probability that the number of challenged blocks that are among the non-missing α fraction of blocks is at least $c\alpha + 1$. The number of possible cases that more than $c\alpha + 1$ challenged blocks are from the non-missing α fraction of the file is: $\binom{n\alpha}{c} + \binom{n\alpha}{c-1} + ... + \binom{n\alpha}{c-c(1-\alpha)+1}$, where n is the total number of file blocks.

Thus, $P = \frac{\binom{n\alpha}{c} + \binom{n\alpha}{c-1} + ... + \binom{n\alpha}{c-c(1-\alpha)+1}}{\binom{n}{c}}$. Considering that $\binom{n\alpha}{x-1} \leq \binom{n\alpha}{x}$ whenever $2 \leq x \leq \frac{n\alpha+1}{2}$, and that $c \leq \frac{n\alpha+1}{2}$ always holds in practice because c is a small constant in the RDC literature (*e.g.*, $c = 400$) compared to n, we have:

$P \leq \frac{\binom{n\alpha}{c}c(1-\alpha)}{\binom{n}{c}} = \frac{\binom{n\alpha}{c}}{\binom{n}{c}}c(1-\alpha) = \frac{n\alpha(n\alpha-1)...(n\alpha-c+1)}{n(n-1)...(n-c+1)}c(1-\alpha) = \frac{n\alpha}{n}\frac{n\alpha-1}{n-1}...\frac{n\alpha-c+1}{n-c+1}c(1-\alpha) = \alpha\frac{n}{n}\alpha\frac{n-\frac{1}{\alpha}}{n-1}...\alpha\frac{n-\frac{c-1}{\alpha}}{n-(c-1)}c(1-\alpha) \leq \alpha^c c(1-\alpha)$.

Thus, $P \leq \alpha^c c(1-\alpha)$. $\square$

7. Implementation and Experiments

7.1 Background on Amazon's Cloud Services (AWS)

We provide some background for Amazon's cloud services within the United States, called Amazon Web Services (AWS). EC2 is Amazon's cloud computing service and S3 is Amazon's cloud storage service. In the United States, Amazon has three EC2 regions (US East - Virginia, US West - North California, and US West - Oregon) and three S3 regions (US Standard, US West - North California, and US West - Oregon). Based on our measurements in Table 2 and 3 of Appendix A, the following EC2 and S3 regions are located extremely close to each other and have very high network connection between them, thus we consider them in the same region: Virginia (EC2 US East - Virginia and S3 US Standard), N. California (EC2 US West - North California and S3 US West - North California), and Oregon (EC2 US West - Oregon and S3 US West - Oregon).

7.2 Experimental Setup

We build and test our prototype for $\mathsf{RDC-SR}$ on Amazon Web Services (AWS). Each server is run on an EC2 large instance (4 ECUs, 2 Cores, and 7.5GB Memory, created from Amazon Linux AMI 64-bit image). The client is run on a machine located in our institute, equipped with Intel Core 2 Duo system with two CPUs (each running at 3.0GHz, with a 6144KB cache), 333GHz frontside bus, 4GB RAM and a Hitachi HDP725032GLA360 360GB hard disk with ext4 file system. In the following, our EC2 instances and S3 data are located in the Oregon region, unless noted otherwise. The prototype for $\mathsf{RDC-SR}$ has been implemented in C and uses OpenSSL version 1.0.0e [1] for cryptographic operations.

From Sec. 5, we have $\eta \geq \frac{x}{(1-\alpha)\cdot s\cdot t_{prf}}$ and we also choose $x = 0.034$ *sec*. We estimate $t_{prf} = 4.3$ μsec for an EC2 large instance (EC2 Oregon). We choose 40 KB for the file block size and 80-bit prime number p, thus s is 4000.

We use the following values for (α, η) in our experiments: $(0.6, 5), (0.7, 7), (0.8, 10)$ (recall from Sec. 5 that once α is fixed, η can be computed). We use these values for α to reflect an economically-motivated CSP (such a CSP would not likely be interested in saving a small amount of storage, so we don't consider cases when $\alpha > 0.8$). The experimental results are averaged over 20 runs, unless noted otherwise.

Preprocess. The file to be outsourced is preprocessed by an EC2 large instance, generating 3 different replicas and the corresponding verification tags. The replicas are then stored at 3 different S3 regions, one replica per region. All the verification tags are stored at every S3 region. In our experiments, we adopt a slightly different strategy from the scheme described in Sec. 4: One of the 3 different replicas is the actual original file. This strategy speeds up the repairing of a corrupted replica, because the replica can be computed directly from the original file (a similar approach was proposed in [13, 19]).

We measure the time for masking, verification tag generation and total preprocessing for one masked replica un-

α	η	operation	throughput (MB/s)
0.6	5	masking	0.44
		verification tag	5.2
		total	0.41
0.7	7	masking	0.32
		verification tag	5.2
		total	0.3
0.8	10	masking	0.22
		verification tag	5.2
		total	0.21

Table 1: Preprocessing throughput

der three sets of (α, η) parameters. We repeat the experiments for four different file sizes (20MB, 50MB, 80MB, and 100MB). Table 1 shows the throughput for total preprocessing and its different components.

We have several observations for Table 1: First, the throughput of masking operation decreases when α increases. This is expected because a larger α means that it is more difficult to detect the adversarial behavior, thus, we need a larger η, hence more computations are required for masking. Secondly, the throughput of verification tag computation is independent of α, due to the fact that the verification tags are computed over the masked replica, which is independent of η, hence independent of α. Thirdly, the throughput of total preprocessing, which includes masking and verification tag computation, is always close to but a little smaller than the throughput of masking, since the verification tag computation is very efficient (can generate verification tags for more than 5MB data in one second) and only has a small impact to the total preprocessing time.

Challenge. The client issues a challenge to the server (run in an EC2 large instance). The server samples blocks from S3 in the same region, and computes and sends back the proof. The client then checks the proof. For simplicity, we only challenge the server running in EC2 Oregon which is responsible for the replica stored in S3 Oregon. The number of blocks to be challenged is $c = 400$, which provides a high guarantee to detect data corruption by the server [3]. For the chosen values of α (Table 1) and c, the probability that a server performs the ROTF attack without being detected is less than $1.38 * 10^{-37}$(cf. Sec. 6). Amazon S3 offers a REST API to access data, which is based on the HTTP/1.0 protocol. Although HTTP supports operations on multiple ranges of the target object in one request, Amazon S3 only supports one range. This means that in order to sample 400 random blocks, we must send 400 different requests for a one-block range. This explains partially the large variation we observe in block access time for S3 (Figures 4(b) and 5(b)), thus we average the block access time over 100 runs. We examine two cases:

- *Benign case:* The CSP is honest, *i.e.*, it strictly stores the replicas in the corresponding regions according to the contract. Upon challenge, the server uses the data from the same region to pass the challenge. In this case, the total server computation includes sampling challenged blocks from S3 of the same region and computing the proof.
- *Adversarial case:* The CSP is cheating by not storing all replicas in their entirety according to the contract. The malicious CSP adopts the best attack strategy described in Sec. 3.2. Because the server will only have an α fraction of the challenged blocks, it retrieves the other $(1-\alpha)$ fraction from another region and recreates the missing blocks on the fly. The total server computation for this case includes sampling challenged blocks from S3 of the same region, generating a $1-\alpha$ fraction of the challenged blocks (by masking the original file blocks), and computing the proof.

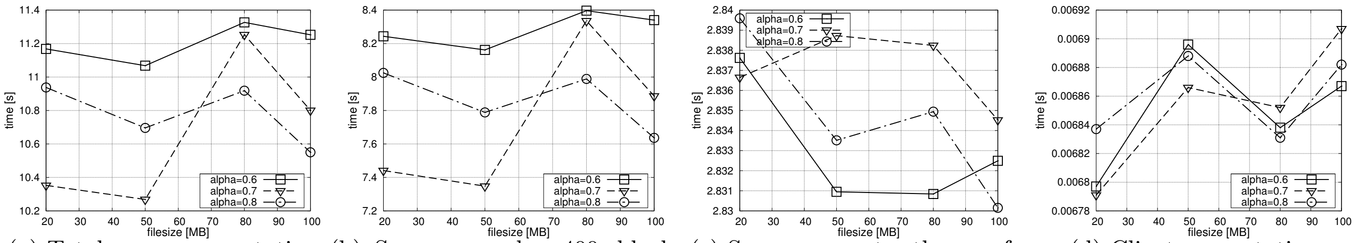
We repeat the experiments for different sets of (α, η) parameters and for different file sizes. Figure 4 and 5 show the server computation and client computation for both cases.

For the benign case, we observe from Fig. 4 that the total server computation and its various components as well as the client (verifier) computation are independent of file size and of α. This is expected because: First of all, we rely on spot checking [3] which always randomly samples a fixed number of blocks from the masked replica, thus can maintain constant server/client computation. Secondly, during a challenge, the operations on both server and client are over the masked replica, which is independent of η, hence independent of α. Figure 4(d) shows that the time for the client to check the proof is less than 7 *msec*, which justifies our claim that the system imposes a small load on the verifier during the challenge phase.

For the adversarial case, we observe from Fig. 5 that the total server computation and its various components are independent of filesize. The reason has been explained in the benign case. For Figure 5(c), we expected to see that the masking time is independent of α, because: The malicious server always stores only an α fraction of the corresponding data, and generates the $1-\alpha$ fraction of challenged blocks on the fly (by masking). Larger α means that the malicious server has to generate less challenged blocks but generating one challenged block will be more expensive, thus, the masking time for the $1-\alpha$ fraction of challenged blocks should be almost constant. However, Figure 5(c) shows that for the case of $\alpha = 0.7$, the masking time is larger than those of other two cases. This discrepancy can be explained because we must always choose η as an integer number. The server masking time is $400(1-\alpha)\cdot s\cdot\eta\cdot t_{prf}$, which is determined by the multiplication of $1-\alpha$ and η. For the case of $\alpha = 0.7$, the minimum integer for η is 7, thus, $(1-\alpha)\cdot\eta = 2.1$. For both cases $\alpha = 0.6$ and $\alpha = 0.8$, $(1-\alpha)\cdot\eta = 2 < 2.1$. This explains where such a discrepancy comes from. However, note that there is a lower bound on the server masking time, because $400(1-\alpha)\cdot s\cdot\eta\cdot t_{prf} \geq 400(1-\alpha)\cdot s\cdot t_{prf}\cdot\frac{x}{(1-\alpha)\cdot s\cdot t_{prf}} = 400x =$ 13.6 *sec*. We observe that most of the points in Figure 5(c) are over this lower bound, except the point in 20MB filesize when $\alpha = 0.6$, but we still consider this point as valid since it is only 1% smaller. The existence of the lower bound for the server masking time guarantees that even if the malicious server has the magic power to access the data and compute the proof instantly (*i.e.*, the times shown in Figure 5(b) and Figure 5(d) are 0), it still cannot cheat successfully, since the time for generating the $1-\alpha$ fraction of challenged blocks will be always larger than $400x$, which is the total server computation for the benign case.

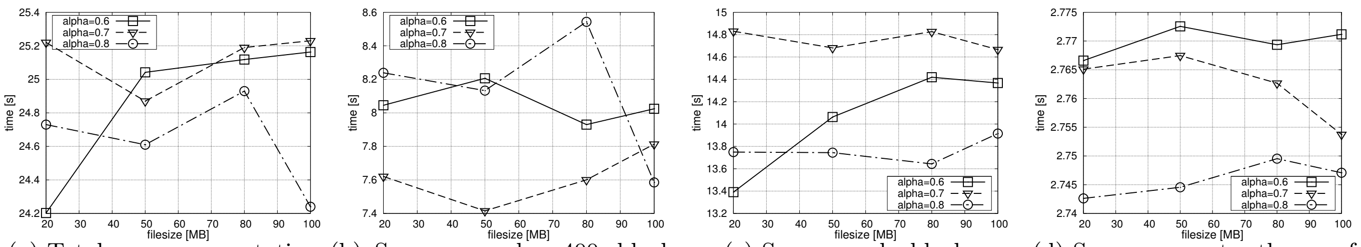
Figure 5(d) shows that the time for the server to compute the proof varies with α. However, we can still conclude that this time is independent of α given that the variance is quite small (around 1%).

According to the guidelines for establishing the time threshold τ in Sec. 5, τ should be 13.7 *sec* (c=400, $x =$ 0.034 *sec*, $t_i = 0.045$ *sec* based on our experiments). We see



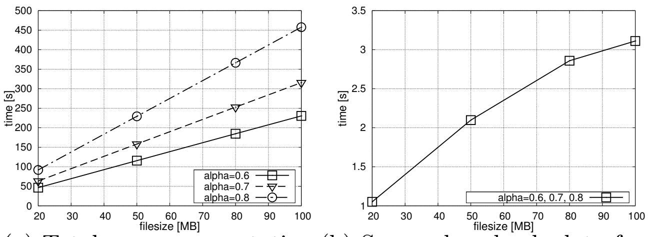
(a) Total server computation (b) Server samples 400 blocks from S3 in the same region (c) Server computes the proof (d) Client computation

Figure 4: Computational cost for the server and the client in challenge phase (benign case).



(a) Total server computation (b) Server samples 400 blocks from S3 in the same region (c) Server masks blocks (d) Server computes the proof

Figure 5: Computational cost for the server and its various components in challenge phase (adversarial case).



(a) Total server computation (b) Server downloads data from another region

Figure 6: Computational cost for repairing a replica.

that 95% of the individual runs for Figure 4(a) are below this threshold, and 100% of the individual runs for Figure 5(a) are above this threshold. This confirms the practical value of using a time threshold to establish if the CSP is malicious.

Repair. We assume that the replica stored in S3 Oregon has been found corrupted, and the replica stored in S3 California is retrieved to repair the corruption. The repair server runs in a large instance from EC2 Oregon. The server downloads the replica from S3 California and masks it to generate the replica for S3 Oregon. The server also downloads all the verification tags from another S3 region (this time is negligible in our experiment). The results are shown in Figure 6 (this includes time for masking to generate a new replica, as described Table 1). We observe from Figure 6(a) that, for repairing one replica, total server computation increases with α. This is because, as shown in Preprocessing, larger α will result in larger masking computation, and the masking computation dominates the total repair computation.

One significant advantage in the repair phase is that the client can be kept lightweight, *e.g.*, the client only needs to exchange a few messages to coordinate the repair procedure. This justifies our claim that the system imposes a small management load on the data owner during repair.

8. Related Work

RDC for the single-server setting. Early RDC schemes have focused on ensuring the integrity of outsourced data in the static setting. Such schemes include Provable Data Possession (PDP) [3] and Proofs of Retrievability (PoR) [15, 21]. Later RDC schemes investigated models that can provide strong integrity guarantees while supporting dynamic operations on the outsourced data [5, 14, 25, 23, 28, 10, 9].

RDCs for the multiple-server setting. RDC has been extended to the multiple-server setting (distributed RDC). Curtmola et al. proposed MR-PDP [12], an efficient RDC scheme for replication-based distributed storage systems, which differentiates the replicas by random masking. We adapt this technique in our work. Bowers et al. [7] and Wang et al. [24] built RDC schemes for erasure coding-based distributed storage systems. Chen et al. [11] proposed an RDC scheme for network coding-based distributed storage systems. All the aforementioned distributed RDCs adopt client-side repair, in which the client is intensively involved in the repair procedure, *i.e.*, the client will retrieve the data, generate and upload the new data to repair the corruption. Our work proposes server-side repair, a novel strategy which is different from all the previous distributed RDCs.

A new direction for RDC. All the previous RDC schemes are cryptography-based, *i.e.*, the security of the proposed schemes are inherited from the security of the cryptographic primitives. Bowers et al. [8] propose RAFT, a new time-based RDC scheme which can enable a client to obtain a proof that a given file is distributed across an expected number of physical storage devices in a single datacenter.

Although RAFT and our work share the idea of using a time-based mechanism to detect malicious behavior, they are fundamentally different in their basic approach and goals, and in the system and adversarial models. *First*, while in $\mathsf{RDC-SR}$ the replicas are differentiated based on controllable masking to mitigate the ROTF attack, RAFT mainly relies on the I/O bottleneck of a single hard drive, specifically, on the fact that the time required for two parallel reads from two different drives is clearly less that the time required for two sequential reads from a single drive. *Second*, in $\mathsf{RDC-SR}$ the file is replicated t times and the t replicas are stored in t different data centers (which may belong to the same CSP or to different CSPs). Within one data center, $\mathsf{RDC-SR}$ does not impose requirements on how exactly should the replica be stored. The data owner seeks to enable the self-repairing functionality while ensuring that a certain number of replicas are stored in the cloud at all times, so that the desired level of reliability is maintained. In RAFT, the file is encoded and is stored by the cloud server using the desired number of hard drives. The data owner wants to ensure that the server stores the file so that it can tolerate a certain number of hard drive failures. *Third*, in $\mathsf{RDC-SR}$ we introduce the α-cheating adversary, in which the cloud servers collude with each other to cheat by only storing an α fraction of the contractual storage, and there are no requirements for how exactly the adversary stores the data on the hard drives. In RAFT, a cheap-and-lazy adversary tries to cut corners by storing less redundant data on a smaller number of disks or by mapping file blocks unevenly across hard drives.

Benson et al. [6] propose another time-based model (BDS model) to guarantee that multiple replicas are distributed to different data centers of the CSP. Our work adapts this model to enable the server-side repair.

Watson et al. [26] propose LoSt, which formalizes the concept of Proofs of Location (PoL). A PoL relies on a geolocation scheme [6] and a Proof of Retrievability (PoR) scheme. We summarize the differences between $\mathsf{RDC-SR}$ and LoSt. *First*, the goals are different. $\mathsf{RDC-SR}$ aims at enabling self-repair, a novel functionality for replication-based distributed storage systems that, when combined with periodic integrity checks provides an efficient mechanism to ensure long-term data reliability. In particular, $\mathsf{RDC-SR}$ does not try to enforce specific locations of the data. LoSt aims at ensuring that the outsourced file copies are stored within the specified region and requires a landmark infrastructure to verify the location of the data. *Second*, the system model is different. $\mathsf{RDC-SR}$ has two entities, namely, the client and the storage servers (CSP), in which the client is always trusted and the storage servers are untrusted and may collude. In LoSt there are three entities, the client, the CSP, and the data centers, and the model assumes that theres is no collusion between the CSP and the data centers. *Third*, the basic idea for the solution is different. $\mathsf{RDC-SR}$ relies on the differentiation of the replicas based on controllable masking to defend against the ROTF (replicate on the fly) attack. Instead, LoSt relies on "recoding" to efficiently differentiate (done at the CSP with CSP's private key) the file tags for each server, while each server will keep the same file copy.

Other work. Similar with the work of Reiter at al. [18], our scheme relies on the idea that only a prover which has the data can respond quickly enough to pass a challenge. Unlike our work however, their work is set in the context of P2P networks and the verifier (client) needs to keep the data for the verification purpose.

9. Acknowledgements

This research was sponsored by the US National Science Foundation CAREER grant 1054754-CNS.

10. References

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

[2] Wget. `http://www.gnu.org/software/wget/`.

[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 Conference on Computer and Communications Security (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 Conference on Computer and Communications Security (CCS '11)*, 2011.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. of ACM Conference on Computer and Communications Security (CCS '09)*, 2009.

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

[16] H. Krawczyk. LFSR-based hashing and authentication. In *Proc. of Annual International Cryptology Conference (CRYPTO '94)*, 1994.

[17] Z. N. J. Peterson, M. Gondree, and R. Beverly. A position paper on data sovereignty: the importance of

geolocating data in the cloud. In *Proc. of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*, 2011.

[18] M. K. Reiter, V. Sekar, C. Spensky, and Z. Zhang. Making peer-assisted content distribution robust to collusion using bandwidth puzzles. *Information Systems Security*, pages 132–147, 2009.

[19] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *Proc. of International workshop on Peer-To-Peer Systems (IPTPS '05)*, 2005.

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

[21] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. of Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '08)*, 2008.

[22] V. Shoup. On fast and provably secure message authentication based on universal hashing. In *Proc. of Annual International Cryptology Conference (CRYPTO '96)*, 1996.

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

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

[25] 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.

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

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

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

APPENDIX

A. Measurements for the Amazon CSP

Tables 2 and 3 show the bandwidth and the propagation delay between Amazon S3 data centers (regions) and between our institution and different S3 data centers (regions). For measurements, we used an EC2 instance within the corresponding Amazon data centers. To measure bandwidth, we used Wget [2] to download a large file. To measure the propagation delay, we adopt the method introduced in [6] that is, we measure the time between sending a SYN packet and receiving a SYN-ACK packet of a TCP connection, half of which is considered as the propagation delay. All the results in Tables 2 and 3 are averaged over 20 runs.

	Virginia	N. California	Oregon
Virginia	32.7	11.62	12.59
N. California	11.95	48.03	36.05
Oregon	14.07	26.43	52.18
Our institution	0.816	0.456	0.439

Table 2: Download bandwidth (in MB/s): (rows 1-3) between S3 data centers (regions); (row 4) between our institution and different S3 data centers.

	Virginia	N. California	Oregon
Virginia	0.579	40	49
N. California	40	0.705	11
Oregon	49	11	0.212
Our institution	4	40	45

Table 3: Propagation delay (in milliseconds): (rows 1-3) between S3 data centers (regions); (row 4) between our institution and different S3 data centers.

B. Network Delays for Amazon CSP

Table 4 shows the values of $t_{ij} + t_j - t_i$ for Amazon AWS, which are used in Sec. 5.

i	j	$t_{ij} + t_j - t_i$ (in seconds)
Virginia	N. California	0.08
Virginia	Oregon	0.098
N. California	Virginia	0.08
N. California	Oregon	0.022
Oregon	Virginia	0.098
Oregon	N. California	0.022

Table 4: Values of $t_{ij} + t_j - t_i$ if the client is located within a data region of AWS S3

C. Sampling Blocks from Amazon S3

We wrote a program running in an EC2 instance (Amazon Virginia region) to randomly sample 4KB blocks from S3 Virginia region. We collect the time in Table 5. All the results are averaged over 20 runs.

# of blocks	1	10	40	400
time (sec.)	0.026062	0.260492	1.024863	10.191946

Table 5: The time for randomly sampling 4KB blocks from Amazon S3 Virginia region