Journal of Computer Security 25 (2017) 537–584
DOI 10.3233/JCS-16868
IOS Press

Remote data integrity checking with server-side repair[1]

Bo Chen [a] and Reza Curtmola [b,*]
[a] *Department of Computer Science, Michigan Technological University, Houghton, MI, USA*
E-mail: bchen1@mtu.edu
[b] *Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, USA*
E-mail: crix@njit.edu

Abstract. Distributed storage systems store data redundantly at multiple servers that 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 work on a new concept, namely, server-side repair, in which the servers are responsible to repair the corruption, whereas the client acts as a lightweight *repair coordinator* during repair. We propose two novel RDC schemes for replication-based distributed storage systems, RDC-SR and ERDC-SR, which enable server-side repair (thus taking advantage of the premium connections available between a CSP's data centers) and minimize the load on the client side. Although both schemes achieve a similar objective, RDC-SR assumes that the computational power of the CSP will not grow over time, whereas ERDC-SR relaxes this assumption and considers a CSP whose computational power can increase over time. Our guidelines on choosing the parameters of these schemes provide insights on their practical usage and also reveal that, whereas ERDC-SR can handle more powerful adversaries, it also imposes a minimal file size.

Finally, we evaluate the performance of the two schemes. For the RDC-SR scheme, we build a prototype on the Amazon cloud and provide experimental results to support its effectiveness. Our prototype for RDC-SR built on Amazon AWS validates the practicality of this new approach. For the ERDC-SR scheme, our analytical performance analysis shows that the scheme is an order of magnitude more efficient than a simple extension of RDC-SR to defend against the stronger adversarial model.

Keywords: Cloud storage, remote data integrity checking, server-side repair, replicate on the fly attack, butterfly encoding

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 that 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 work however, we consider a setting in which the storage servers are untrusted and may act maliciously. In this setting, Remote Data

[1]A preliminary version of this article was published in [13].
[*]Corresponding author. E-mail: crix@njit.edu.

Checking (RDC) [4,5,27,34] 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, Challenge, and Repair. To outsource a file 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 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 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 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 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 [8,16], 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 ([8,16]). 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).

We consider a new storage system architecture in which each storage server exposes an interface for data manipulation so that the data owner can coordinate the actions of the storage servers in the Repair phase. To repair a faulty server during Repair, the data owner identifies healthy servers and instructs them to collaborate. In this process, most of the communication occurs between the storage servers, and the communication between data owner and storage servers is minimized.

Main objective: Informally, our main objective is to design an RDC scheme for a replication-based distributed storage system that is based on untrusted servers and achieves 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 ([16] provides multiple replica guarantees, whereas RDC based on spot-checking [4,27,34] 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 for checking replica storage and by redesigning 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 [7,30] 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 Section 2.1 we show that this approach has major limitations. Moreover, we show that for real-world CSPs, one of the assumptions made by prior work [7] 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 8 (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 ([16]) 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 work, we propose to use a different paradigm, in which the data owner gives the servers both access to the original file[2] and the means to generate new replicas. This will allow the servers to generate a new replica by collaborating between themselves during Repair. Note that if the confidentiality of the outsourced data is a concern, replicas can be encrypted and our approaches can be immediately applied to the encrypted data.

A basic approach and its limitations. A straightforward approach would be for the data owner to create *different* replicas by encrypting the original file. The data owner would reveal to the servers the

[2]If data confidentiality is a concern, data can be first encrypted and our approach can be applied on top of the encrypted data.

key material used to create the 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). Whenever 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 the desired t replicas, unbeknownst to the client.

Overcoming the ROTF attack. The new paradigm we introduce in this work, 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.

We consider two types of economically-motivated adversaries, a *static adversary* and a *dynamic adversary*. A static adversary refers to an adversary who performs the ROTF attack by using a fixed amount of computational power. This captures a CSP who initially has a fixed budget for its computational power, and does not increase its budget over time. Our first scheme, RDC-SR, defends against this type of static adversary by using a controllable amount of masking, *i.e.*, a technique that combines the original file with randomness in order to obtain the different replicas to be stored at different servers. The computational load induced by the masking operation can be variable and is decided by the data owner. In practical applications, however, the CSP may increase its budget to purchase more computational power, *i.e.*, the adversary may perform the ROTF attack based on an increased computational power over time. RDC-SR, unfortunately, cannot defend against this type of dynamic adversary. One alternative to address a dynamic adversary is to extend RDC-SR by simply increasing the amount of masking to match the adversary's computational power at the end of the considered time interval (we call this scheme SRDC-SR). This approach, however, is inefficient because it will lead to expensive Setup and Repair phases. Instead, we propose our second scheme, ERDC-SR, in which we create dependencies between each of the replica blocks and multiple original file blocks. Compared to SRDC-SR, ERDC-SR introduces much less overhead in both the Setup and the Repair phases in order to defend against a dynamic adversary.

A comparison between RDC-SR and ERDC-SR. Both RDC-SR and ERDC-SR are RDC schemes proposed to support server-side repair for replication-based distributed storage systems. Although they share the same overall goal, they are different in several aspects. First of all, both schemes allow server-side repair, and mitigate the ROTF attack performed by the economically-motivated malicious CSP. However, RDC-SR can only defend against an adversary who has a fixed amount of computational power, and ERDC-SR can defend against a stronger adversary that can increase its computational power over time. Secondly, during the Setup phase, both schemes preprocess the original file, generating different replicas and the corresponding metadata. However, to create a replica, RDC-SR relies on a controllable amount of masking, whereas ERDC-SR utilizes a variant of butterfly encoding [41]. Thirdly, in RDC-SR, each replica block depends on only one original file block. However, in ERDC-SR, each replica block depends on multiple original file blocks. Thus, during its Challenge phase, to answer a challenge by performing the ROTF attack, the adversary has to compute not only the challenged replica blocks that are missing, but also many other intermediate blocks needed to compute these challenged blocks. Lastly, RDC-SR can be applied to files of arbitrary sizes, whereas ERDC-SR can only be applied to files over a certain size (*e.g.*, under typical parameters, a file needs to be at least 60 MB for a target protection interval of 2 years, and at least 220 MB for an interval of 5 years).

Contributions. In this paper, we propose two novel RDC schemes for replication-based distributed storage systems, RDC-SR and ERDC-SR, which enable 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 schemes impose 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 RDC schemes that implement 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 (Section 2.1).
- 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. Our new model for checking replica storage allows servers to generate new replicas and shifts the burden during the Repair phase to the server side, allowing lightweight clients to perform data maintenance (Section 2.2). 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. We prove that the best data distribution strategy for an α-cheating adversary is to store in each of the storage servers an equal fraction of the whole contractual storage (Section 3).
- 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 (Section 4.1). 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 accommodate our new model, 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 performed by a static α-cheating adversary (Section 4.3).
- We propose ERDC-SR, another novel RDC scheme for replication-based distributed storage systems, which can support server-side repair (Section 5). Compared to RDC-SR, ERDC-SR can defend against a dynamic α-cheating adversary that can increase its computational power over time. We use a butterfly encoding to create dependencies between each replica block and multiple original file blocks. As a result, an adversary who wants to pass the challenge by performing the ROTF attack, needs to compute on the fly not only the challenged replica blocks that are missing, but also many intermediate blocks needed to compute these challenged blocks. We prove that ERDC-SR can mitigate the ROTF attack performed by a dynamic α-cheating adversary (Section 5.3).

- We provide guidelines on how to choose the parameters for RDC-SR in practical settings (Section 4.2) 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, whereas 100% of the adversarial cases are over the threshold (Section 6.1).
- For ERDC-SR, we provide an instantiation for the cryptographic transformation used in the butterfly encoding (Section 5.2.1). We also derive guidelines on how to choose the scheme's parameters in practical settings, and observe that ERDC-SR imposes a minimal filesize under typical parameters (Section 5.2.2). Additionally, we provide analytical performance comparisons between ERDC-SR and SRDC-SR. Our analysis shows that, for both Setup and Repair, the overall computation of ERDC-SR is an order of magnitude less than that of SRDC-SR (Section 6.2).

Structure of the paper. The paper is organized as follows: In Section 2, we review a network delay-based model and point out its limitations. We then revise this model and include replica differentiation and time-consuming replica generation in order to limit the adversary's ability to cheat. In Section 3, we introduce our system and adversarial model, as well as adversarial strategies. In Section 4, we introduce RDC-SR, an RDC scheme for replication-based distributed storage systems, which can support server-side repair and defend against a static α-cheating adversary. In Section 5, we introduce ERDC-SR, another RDC scheme for replication-based distributed storage systems, which can defend against a dynamic α-cheating adversary. We provide an experimental evaluation for RDC-SR and an analytical performance analysis for ERDC-SR in Section 6. We then give an overview of the related work in Section 7 and conclude in Section 8.

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 different geographic locations [7] (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 [7]. 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

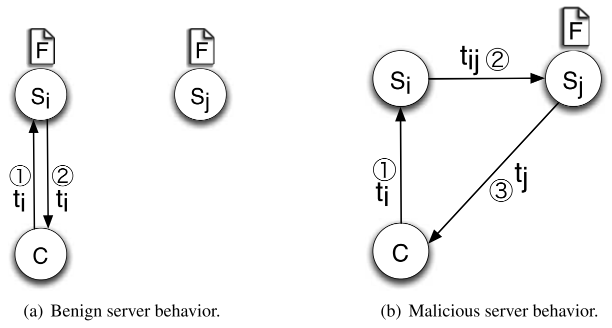


Fig. 1. Auditing protocol: Client C checks if server S_i has a file copy F.

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 data center 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 Fig. 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 data center 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, $T_i + 2 * \max(t_i) \leqslant \min(t_i) + \min(t_{ij}) + \min(t_j)$, or

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

Limitations of the basic PoR protocol based on BDS model. Based on the condition derived from the BDS model, Benson et al. [7] 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 8 and 9 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 8, 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 9 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 8 and 9, 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 4 KB random file blocks from S_i, Equation (1) for the basic PoR protocol becomes $x \cdot k \leqslant 80 + 0.3k$, 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 B), thus $k \leqslant 2.66$. 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 [4]).

2.2. *A new model to enable server-side repair*

The main problem with the basic PoR protocol based on the BDS model (cf. Section 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:

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

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 data centers of the cloud provider are known.
(Assumption 2) For each data center 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 $\mathtt{F}$. 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 $\mathtt{F}$ from $\mathtt{F}_i$, and then generate $\mathtt{F}_j$.

The file $\mathtt{F}$ has n blocks and $|\mathtt{F}|$ denotes the size of $\mathtt{F}$ in bytes.

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 the economic benefit of 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 [9,16].

3.2.1. The ROTF attack

To illustrate the ROTF attack that was introduced in Section 1, consider the setting in Fig. 1(b), where S_i and S_j should 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.

3.2.2. 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 $\frac{1}{t} \leqslant \alpha \leqslant 1$).

Note that if $\alpha < \frac{1}{t}$, then the CSP stores less than $|\mathtt{F}|$, which means that any single-replica RDC scheme [4,27] would be enough to detect the CSP's dishonest behavior. Thus, we do not consider the case when $\alpha < \frac{1}{t}$.

We consider two types of α-cheating adversaries. A **static α-cheating adversary** refers to a CSP that possesses a fixed amount of computational power that is known (or can be estimated) by the client, and does not increase its computational power over time. This captures a setting in which the CSP initially has a fixed budget for its computational power, and does not increase its budget over time. We then relax this assumption about the adversary's computational capabilities by allowing the CSP to grow its computational power over time, and consider a dynamic α-cheating adversary. A **dynamic α-cheating adversary** refers to a CSP that initially has a known amount of computational power and is capable of increasing its computational power at a known rate over time. Compared to the static α-cheating adversary, this captures a more flexible and realistic setting, in which the CSP is allowed to increase the budget for its computational power according to its own business strategy. We argue that it is reasonable to assume the CSP increases its computational power at a growth rate known by the client because: first of all, a rational and economically-motivated CSP will increase its budget steadily, rather than arbitrarily. Thus, it is possible for the client to estimate how the CSP will grow the budget for its computational power based on the CSP's historical records; secondly, the unit price of the computational power usually decreases steadily, and the client can estimate how this price will decrease based on the public historical records (*e.g.*, the historical price of processor and memory). We propose solutions that can successfully defend against static α-cheating adversaries (Section 4) and against dynamic α-cheating adversaries (Section 5).

3.2.3. 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 to pass challenges is always better off by keeping a copy of the original file $\mathtt{F}$. 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 $\mathtt{F}$ 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 $\mathtt{F}$ from an existing replica, say $\mathtt{F}_j$, and then generate $\mathtt{F}_i$ from $\mathtt{F}$. Since replica generation is a time consuming operation (and similarly recovering $\mathtt{F}$ 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 $\mathtt{F}$.

Also, recall that most RDC schemes ensure efficiency by using spot checking [4,27,34]: 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.

The data distribution strategy for an α-cheating adversary. 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 cheat and remain undetected. A *basic strategy* is when the adversary chooses to store on one of the servers the original file $\mathtt{F}$, and to store on each of $\lfloor \alpha t \rfloor - 1$ servers an amount of data equal to the size of a replica (*e.g.*, each of these servers stores its corresponding 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 always needs to generate the c challenged blocks on the fly and then construct the answer to the challenge.

Theorem 3.2 shows that the *best data distribution strategy for an α-cheating adversary* that wants to remain undetected is to store in each of the t servers an equal fraction of the whole storage, *i.e.*, $\alpha|F|$ bits at each server. Thus, the adversary will still only use $\alpha t|\mathtt{F}|$ storage space in total.[3] 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 (since this server stores an α fraction of the corresponding replica). Thus, on average, it only needs to generate on the fly an $(1 - \alpha)$ fraction of the c challenged blocks.

Theorem 3.2. *For an α-cheating adversary, the best data distribution strategy to remain undetected is to store in each of the t servers an equal fraction of the whole $\alpha t|\mathtt{F}|$ storage.*

The proof is provided in Appendix C.

4. An RDC scheme with server-side repair

In this section, we propose RDC-SR, the first replication-based Remote Data Checking scheme that supports Server-side Repair and defends against a static α-cheating adversary.

4.1. RDC-SR

The original file $\mathtt{F}$ has n blocks, $\mathtt{F} = \{\mathtt{b}_1, \dots, \mathtt{b}_n\}$, 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.*, $j \in \{1, \dots, n\}$), and k to denote the index of a symbol in a block (*i.e.*, $k \in \{1, \dots, s\}$). Let κ be a security parameter. We make use of two pseudo-random functions (PRFs) h and γ with the following parameters:

– $h : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^{\log p}$

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

- n: the number of RDC blocks in a file $\mathtt{F}$.
- c: the number of RDC blocks in a replica checked by the verifier during Challenge.
- α: a parameter representing the adversarial strength, *i.e.*, an α-cheating adversary will only store an α fraction of the contractual storage.
- τ: the time threshold, which is used to measure the response time during Challenge.

Fig. 2. A reference sheet with various parameters of RDC-SR.

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

Figure 2 provides a reference sheet with various parameters of RDC-SR.

RDC-SR **overview.** Like any RDC system for a multiple-server setting [8,15,16,42], 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.*, $i \in \{1,\ldots,t\}$). To differentiate the replicas, we adopt a masking strategy similar as in [16], 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 strengths (see Section 3.2). The client then generates verification tags for every replica, one tag per file block. Each verification tag is computed similarly as in [34], namely as a message authentication code by combining universal hashing with a PRF [28,33,37,46]. 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 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 challenges all the storage servers simultaneously, so that a server, who does not store a replica honestly, cannot use other servers' computational power to compute a proof to answer the challenge. For each challenge, the client uses spot checking to check the replica at that server, in which it randomly samples a small subset of blocks from the corresponding replica and checks their validity based on the server's response. Such a technique can detect replica corruption with high probability [4], and has the advantage of only imposing a small overhead on both the client and the server. We use a time threshold τ for our new model (see Section 2.2): If the response from a server is not received within time τ, then that replica will be considered corrupted. Since a static α-cheating CSP possesses a fixed amount of computational power known by the client, it is possible for the client to estimate τ.

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 (as shown in Table 8 of Appendix A) and allow the servers to collaborate among themselves to restore the corrupted replica. This is possible because 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 Figs 3 and 4, 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. K_2 will be used to generate distinct replicas and its value is made public by the client. It then picks s

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

1. Run $(\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}, \mathtt{F}_i) \leftarrow \text{GenReplicaAndMetadata}(K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s, \eta)$
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 $\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}$ to each server .

The client may now delete the file $\mathtt{F}$ and stores only a small, constant, amount of data: $K_1, \delta_1, \ldots, \delta_s, \eta$, and τ. K_2 is made public.
Challenge: Client C simultaneously challenges all the storage servers, using spot checking to check possession of each replica stored at each server. In this process, each server uses its stored replica and the corresponding verification tags to prove data possession. As an example, we show the process of challenging 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 $(\rho_1, \ldots, \rho_s, \mathtt{z}) \leftarrow \text{GenProof}(Q, \mathtt{F}_i, \mathtt{z}_{i1}, \ldots, \mathtt{z}_{in})$
3. S_i sends to C the proof of possession $(\rho_1, \ldots, \rho_s, \mathtt{z})$
4. C checks whether the response time is larger than or equal to τ. If yes, C declares S_i as faulty. Otherwise, C checks the validity of the proof $(\rho_1, \ldots, \rho_s, \mathtt{z})$ by running $\text{CheckProof}(K_1, \delta_1, \ldots, \delta_s, Q, \rho_1, \ldots, \rho_s, \mathtt{z}, i)$

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

Fig. 3. RDC-SR: a replication-based RDC system with Server-side Repair.

random numbers, η, and threshold τ. We provide guidelines on how to determine η and τ in Section 4.2. 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 ($l \in \{1, \ldots, \eta\}$). The size of each verification tag is equal to the size of a data symbol, as both are elements in GF(p).

```
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 $1 \leqslant j \leqslant n$:
     • Mask block $\mathtt{b}_j$ at the symbol level and get $\mathtt{m}_{ij}$ (note that $\eta$ denotes the number of masking operations
       imposed on each symbol when generating a distinct replica):
       For $1 \leqslant k \leqslant s$: $\mathtt{m}_{ijk} = \mathtt{b}_{jk} + \sum_{l=1}^{\eta} \gamma_{K_2}(i \parallel j \parallel k \parallel l)$
3. Compute verification tags:
   For $1 \leqslant j \leqslant n$: $\mathtt{z}_{ij} = h_{K_1}(i \parallel j) + \sum_{k=1}^{s} \delta_k \mathtt{m}_{ijk}$
4. Return $(\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}, \mathtt{F}_i = \{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\})$

GenProof($Q, \mathtt{F}_i, \mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}$):
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 $\mathtt{z}$:
     • For $1 \leqslant k \leqslant s$: $\rho_k = \sum_{(j,v_j)\in Q} v_j \mathtt{m}_{ijk} \bmod p$
     • $\mathtt{z} = \sum_{(j,v_j)\in Q} v_j \mathtt{z}_{ij} \bmod p$
3. Return $(\rho_1, \ldots, \rho_s, \mathtt{z})$

CheckProof($K_1, \delta_1, \ldots, \delta_s, Q, \rho_1, \ldots, \rho_s, \mathtt{z}, i$):
1. Parse $Q$ as a set of $c$ pairs $(j, v_j)$
2. If $\mathtt{z} = \sum_{(j,v_j)\in Q} v_j h_{K_1}(i \parallel j) + \sum_{k=1}^{s} \delta_k \rho_k \bmod p$, return "success". Otherwise return "failure".
```

Fig. 4. Components of RDC-SR: KeyGen, GenReplicaAndMetadata, GenProof, CheckProof.

The Challenge phase. For this phase, we integrate spot checking [4,27,34] with our new model for checking replica storage (introduced in Section 2.2). The client (verifier) simultaneously sends a challenge request to 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 [8,15,16]. 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, unmasks 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 verification tags is always small compared to the data.

A concrete example of using RDC-SR. For a concrete example of using RDC-SR, we consider a 4 GB file $\mathtt{F}$ which has $n = 100{,}000$ 40 KB blocks. Each symbol in a block is in GF(p), in which p is an 80-bit prime number, thus, a block should contain $s = 4000$ symbols. The client C wants to outsource this file to $t = 10$ different data centers of the CSP. C considers a security parameter κ with a typical length of 80 bits [35], and randomly picks two keys K_1 and K_2 from GF(2^κ). C then chooses 4000 random numbers $\delta_1, \ldots, \delta_{4000}$ from GF(p). C also chooses $\alpha = 0.8$, and determines the value η and τ according to the guidelines in Section 4.2. During Setup, C generates 10 replicas $\mathtt{F}_1, \ldots, \mathtt{F}_{10}$ by masking the original file $\mathtt{F}$ in symbol level. Each symbol in the original file is masked by η pseudo-random values, which are generated by applying the PRF γ (C uses HMAC for γ, and uses K_2 as HMAC's key) over the concatenation of the replica index i ($i \in \{1, \ldots, 10\}$), the block index j ($j \in \{1, \ldots, 100{,}000\}$), the symbol index k ($k \in \{1, \ldots, 4000\}$) and an integer l ($l \in \{1, \ldots, \eta\}$). For each replica, C generates the set of verification tags, in which C uses HMAC for h, and uses K_1 as HMAC's key. C stores in each data center a replica and the entire set of verification tags. The verification tags require additional storage of 10 MB in each of the 10 data centers. C stores locally K_1, $\delta_1, \ldots, \delta_{4000}$, η and τ, which require around 40 KB storage (each of $\delta_1, \ldots, \delta_{4000}$ is 10 *bytes*; both η and τ are small numbers, each of which requires less than 10 *bytes*; K_1 is 10 *bytes*). C makes K_2 public. During Challenge, C sends to each of the 10 servers a challenge request. Assuming that S deletes 1% of the stored replica, then C can detect server misbehavior with probability over 99% by asking proof for $c = 460$ randomly selected blocks [4]. Each challenge request contains the 460-element set $\{(j, v_j) | j \in \mathbb{Z} \wedge 1 \leqslant j \leqslant 460\}$, which totals around 5 KB (j is an integer smaller than 100,000, v_j is 10 *bytes*). Each server's response contains values $\rho_1, \ldots, \rho_{4000}$, $\mathtt{z}$, which total around 40 KB (each of $\rho_1, \ldots, \rho_{4000}$, t is 10 *bytes*). The client checks the response time with τ, as well as validates the proof of data possession. During Repair, the client coordinates the servers to repair the corrupted replicas, in which the client only needs to send some small coordination messages.

4.2. *Guidelines for using* 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). In the following, we first describe the best adversarial storage strategy in RDC-SR, and then provide guidelines on how to estimate the parameter η and τ in practical applications.

The best storage strategy for the adversary in RDC-SR**.** According to Theorem 3.2, the best data distribution strategy for an α-cheating CSP is to store in each of the t servers an equal fraction of the whole $\alpha t|\mathtt{F}|$ storage, *i.e.*, each server should store $n\alpha$ blocks. For RDC-SR, server i can choose to store blocks of the original file, or blocks of the corresponding replica i, or blocks of other replicas. When a server stores blocks from its corresponding replica, it is able to minimize the time needed to compute a proof of data possession. We conclude that *the best data distribution strategy for an α-cheating adversary in* RDC-SR *is to store in each of the t servers an α fraction of the blocks of the corresponding replica for that server.*

Table 1

Values of $t_{ij} + t_j - t_i$ if the client is located in an AWS S3 region

i	j	$t_{ij} + t_j - t_i$ (in seconds)
Virginia	N. California	0.08
Virginia	Oregon	0.088
N. California	Virginia	0.08
N. California	Oregon	0.021
Oregon	Virginia	0.088
Oregon	N. California	0.022

Estimating η. From Section 2, we have $T_i \leqslant \min(t_i) + \min(t_{ij}) + \min(t_j) + \min(t_R) - 2 * \max(t_i)$, which can be further simplified as $T_i \leqslant t_{ij} + t_j - t_i + t_R$. Let x be the average time each of the c challenged file blocks contributes to the generation of the proof by the server. By knowing the CSP's computational power, the client can estimate x. Since T_i is the upper bound on the execution time of the auditing protocol (as defined in Section 2.1), we have $c \cdot x \leqslant T_i$. Based on the triangle inequality, we always have $t_{ij} + t_j - t_i \geqslant 0$. 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 1, which contains some typical values based on our experiments for Amazon S3). Thus, we get $c \cdot x \leqslant t_R$.

Let t_{prf} denote the time required to compute one PRF (specifically, one computation of the function γ used to mask a symbol in RDC-SR). t_{prf} can be estimated by creating a virtual machine within the CSP's data center, performing a certain number of PRF operations, and averaging the overall computational time. Then, for a challenge that checks c blocks, assuming that the adversary adopts the best storage strategy which is mentioned earlier in this section, we have $t_R = (1 - \alpha) \cdot c \cdot s \cdot \eta \cdot t_{\text{prf}}$. We thus get $c \cdot x \leqslant t_R = (1 - \alpha) \cdot c \cdot s \cdot \eta \cdot t_{\text{prf}}$, which means that $\eta \geqslant \frac{x}{(1-\alpha) \cdot s \cdot t_{\text{prf}}}$ (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 "data access to estimate x" and "regular data access", thus it cannot influence the outcome of verification by artificially manipulating the value of x.

4.3. *Security analysis for* RDC-SR

Our 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, RDC-SR provides the data owner with a guarantee of data possession of that replica by using an efficient spot checking mechanism [4,27]. Note that confidentiality of the data from the CSP is an orthogonal problem to RDC (although our RDC-SR scheme could easily achieve confidentiality by encrypting the original file and then storing masked replicas of the encrypted file).

Different from 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. Theorem 4.2 shows that RDC-SR can mitigate the ROTF attack executed by an α-cheating adversary (defined in Section 3.2):

Lemma 4.1. *In* RDC-SR, *by choosing the parameters η and τ according to the guidelines in Section 4.2, a cheating server who stores an α-fraction of its corresponding replica cannot generate $(1-\alpha)c$ missing blocks within time τ based on its own computational power (where c is the number of file blocks checked by the client in a challenge).*

Proof. According to Section 4.2, $\eta \geqslant \frac{x}{(1-\alpha)\cdot s\cdot t_{\mathrm{prf}}}$, *i.e.*, $\eta \cdot s \cdot t_{\mathrm{prf}} \cdot (1-\alpha)\cdot c \geqslant c \cdot x$. Since τ is computed as $c \cdot x + 2 \cdot t_i$ (Section 4.2), which is approximately $c \cdot x$, considering t_i is negligibly small compared to $c \cdot x$. Thus, $\eta \cdot s \cdot t_{\mathrm{prf}} \cdot (1-\alpha)\cdot c \geqslant \tau$, *i.e.*, to generate the $(1-\alpha)c$ missing blocks, the cheating server at least needs $\eta \cdot s \cdot t_{\mathrm{prf}} \cdot (1-\alpha)\cdot c$ computation, which cannot be done within τ based on her own computational power. □

Theorem 4.2. *In* RDC-SR, *by choosing the parameters η and τ according to the guidelines in Section 4.2, a static 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 the 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 $1/t \leqslant \alpha \leqslant 1$). We have established in Section 4.2 that the best adversarial storage strategy for RDC-SR 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 Section 4.2, the time threshold τ in 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 challenged 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 E be the event that the cheating server is able to cheat successfully without being detected. In RDC-SR, the client challenges all the storage servers simultaneously, and a cheating server cannot use other servers' computational power to compute a data possession proof for the challenged replica. Thus, event E happens when either (a) less than $(1-\alpha)c$ blocks are challenged among the file blocks that are missing at the server (event E_1), or (b) at least $(1-\alpha)c$ blocks are challenged among the missing file blocks but the cheating server is able to generate these missing challenged blocks and compute a proof to answer a challenge within time τ (event E_2). We compute the probability of E as $P(E) = P(E_1) + P(E_2)$.

According to Lemma 4.1, if we choose the parameters η and τ according to the guidelines in Section 4.2, the cheating server cannot generate the missing $(1-\alpha)c$ blocks within time τ. Thus, by choosing η and τ appropriately, we can ensure that event E_2 never happens, so $P(E_2) = 0$.

Evaluating $P(E_1)$ 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: $\binom{n\alpha}{c} + \binom{n\alpha}{c-1} + \cdots + \binom{n\alpha}{c-c(1-\alpha)+1}$, where n is the total number of file blocks.

Thus, $P(E_1) = \frac{\binom{n\alpha}{c}+\binom{n\alpha}{c-1}+\cdots+\binom{n\alpha}{c-c(1-\alpha)+1}}{\binom{n}{c}}$. Considering that $\binom{n\alpha}{x-1} \leqslant \binom{n\alpha}{x}$ whenever $2 \leqslant x \leqslant \frac{n\alpha+1}{2}$, and that $c \leqslant \frac{n\alpha+1}{2}$ always holds in practice because c is a small constant in the RDC literature (*e.g.*, $c = 400$) compared to n, we have:

$$\begin{aligned}
P(E_1) &\leqslant \frac{\binom{n\alpha}{c}c(1-\alpha)}{\binom{n}{c}} = \frac{\binom{n\alpha}{c}}{\binom{n}{c}}c(1-\alpha) = \frac{n\alpha(n\alpha-1)\cdots(n\alpha-c+1)}{n(n-1)\cdots(n-c+1)}c(1-\alpha)\\
&= \frac{n\alpha}{n}\frac{n\alpha-1}{n-1}\cdots\frac{n\alpha-c+1}{n-c+1}c(1-\alpha) = \alpha\frac{n}{n}\alpha\frac{n-\frac{1}{\alpha}}{n-1}\cdots\alpha\frac{n-\frac{c-1}{\alpha}}{n-(c-1)}c(1-\alpha)\\
&\leqslant \alpha^c c(1-\alpha).
\end{aligned}$$

Thus, $P(E_1) \leqslant \alpha^c c(1-\alpha)$, and so $P(E) \leqslant \alpha^c c(1-\alpha)$. $\square$

5. An enhanced RDC scheme with server-side repair

RDC-SR can successfully defend against an α-cheating CSP that does not increase its computational power over time (*i.e.*, a static α-cheating adversary). However, the α-cheating CSP may increase the budget for its computational power. RDC-SR cannot defend against this type of dynamic α-cheating adversary because the time threshold τ is fixed during Setup based on the adversary's estimated computational power. Thus, if the CSP upgrades its computational power after the Setup phase, it may be able to perform the ROTF attack to pass the verification without being detected. Moreover, the client cannot dynamically adjust τ over time after Setup, since it does not know when will the CSP upgrade its computational power. Defending against the dynamic α-cheating adversary is advantageous: After the client has outsourced the storage of a file F, it can obtain an assurance of the reliability and fault tolerance of the data, even if the CSP's computational power increases.

A straw man solution (SRDC-SR) and our enhanced scheme (ERDC-SR). To defend against the dynamic α-cheating CSP, one can simply extend RDC-SR as follows: During Setup, the client first picks η according to the CSP's initial computational power, the growth rate of its computational power, and the length of a time period during which the client wants to obtain an assurance of the reliability and fault tolerance of the outsourced data; it then picks τ and preprocesses the file according to RDC-SR. During

Challenge, the client simultaneously challenges each storage server, requiring each server to prove data possession of a random subset of blocks from its stored replica. For the chosen time period, η is large enough such that an α-cheating server cannot generate the missing challenged blocks on the fly and compute a proof to answer the challenge within τ. The resulting scheme, SRDC-SR, can successfully defend against the dynamic α-cheating adversary. However, it will lead to a much more expensive Setup and Repair phase, since creating a replica will become much more expensive. We thus propose ERDC-SR (Enhanced RDC-SR), in which we enhance RDC-SR by creating dependencies between each of the replica blocks and multiple original file blocks. We show in Section 6.2 that ERDC-SR has a much less expensive Setup and Repair phase than SRDC-SR.

The design of ERDC-SR is based on two main ideas. First, we make each replica block depend on multiple original file blocks, such that computing a replica block on the fly is time-consuming. In order to pass a challenge by performing the ROTF attack, an economically-motivated dishonest CSP needs to compute on the fly not only the challenged replica blocks that are missing, but also many other blocks needed to compute these challenged blocks (*i.e.*, intermediate blocks). Secondly, we determine both the time threshold τ and the amount of work needed to create a replica block based on the CSP's initial computational power and the CSP's computational growth rate. This can ensure that a cheating CSP is not able to compute the intermediate blocks and hence the challenged blocks within time τ when answering a challenge.

5.1. *Design for* ERDC-SR

In ERDC-SR, we make each replica block depend on a subset of β original file blocks. We term β as the dependency factor. A butterfly network [41] was previously proposed to create dependency between an encoded block and all the original blocks in a file. By tuning this technique, we can create dependency between an encoded block and an arbitrary number of original file blocks. In the following, we design a β-butterfly encoding and then propose an ERDC-SR scheme based on β-butterfly encoding. Figure 5 contains a reference sheet with various parameters of ERDC-SR used in this section.

5.1.1. *β-Butterfly encoding*

As shown in Fig. 6, to create a new replica we use the collection of original file blocks as input (at level 0), and apply an atomic *cryptographic transformation* to pairs of blocks in a sequence of $\log\beta$ levels, in which the collection of blocks at level j is the output of a function (*i.e.*, cryptographic transformations over pairs of blocks) over level $j-1$, where $1 \leqslant j \leqslant \log\beta$. The cryptographic transformation used in Fig. 6 takes two input blocks and generates two output blocks such that two properties are satisfied:

– each bit of the pair of output blocks depends on each bit of the pair of input blocks.
– each output block has the same size as the input block.

- n: the number of RDC blocks in a file $\mathtt{F}$.
- c: the number of RDC blocks in a replica checked by the verifier during Challenge.
- α: a parameter representing the adversarial strength, *i.e.*, an α-cheating adversary will only store an α fraction of the contractual storage.
- τ: the time threshold, which is used to measure the response time during Challenge.
- β: dependency factor, defined as the number of original file blocks each replica block depends on, which is equivalent to the number of levels in a butterfly network.

Fig. 5. A reference sheet with various parameters of ERDC-SR.

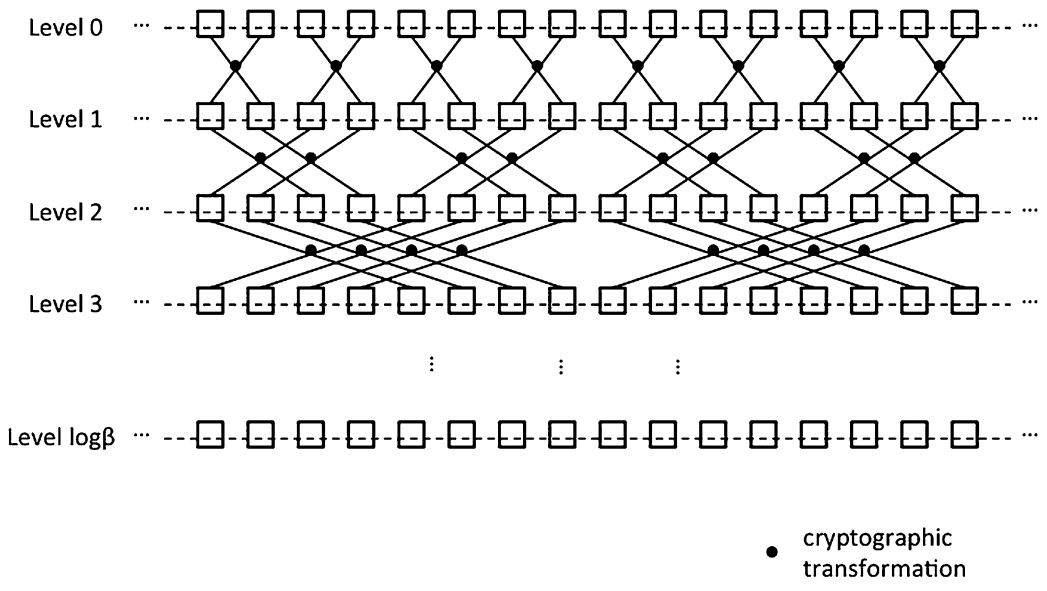


Fig. 6. β-butterfly encoding.

In Section 5.2.1, we provide an instantiation for the cryptographic transformation used in ERDC-SR. Compared to the butterfly encoding used in [41], in which each of the resulting blocks depends on all the original file blocks, the β-butterfly encoding offers more flexibility: each resulting block (at level $\log\beta$) depends on β original file blocks (*i.e.*, blocks on level 0), where $1 \leqslant \beta \leqslant n$. β determines the amount of dependency introduced between the replica blocks and the original file blocks. In Section 5.2.2, we provide guidelines to choose β such that for a certain period of time after the data has been outsourced, a dynamic α-cheating adversary cannot utilize additional computational power to generate the missing data on the fly in order to pass a verification within time threshold τ.

5.1.2. ERDC-SR

In the following, we present the ERDC-SR scheme, in which we use the β-butterfly encoding to create different replicas. The original file $\mathtt{F}$ 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 a security parameter. Let B denote the blocksize. Let h and f be two PRFs, E be the previously described cryptographic transformation, and D be the reverse operation of the cryptographic transformation. h, f, E and D have the following parameters:

- $h : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^{\log p}$
- $f : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$
- $E : \{0,1\}^\kappa \times \{0,1\}^B \times \{0,1\}^B \to \{0,1\}^B \times \{0,1\}^B$
- $D : \{0,1\}^\kappa \times \{0,1\}^B \times \{0,1\}^B \to \{0,1\}^B \times \{0,1\}^B$

ERDC-SR **overview.** During the Setup phase, the client first generates two keys, K_1 and K_2. It then preprocesses the original file, creating t distinct replicas. To create a replica, the client generates a key specific for this replica based on K_2 and the replica index, and applies the β-butterfly encoding to the original file with this key. For each replica, the client computes a set of verification tags based on key

We construct ERDC-SR in three phases, Setup, Challenge and Repair by utilizing the components in both Fig. 4 and 8. 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 $\delta_1, \ldots, \delta_s$ from GF(p). The client also chooses α and determines τ, and then executes:

1. Compute dependency factor β according to the guidelines in Section 5.2
2. Compute and outsource the replicas and the verification tags
   For $1 \leqslant i \leqslant t$:
   - Run $(\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}, \mathtt{F}_i) \leftarrow \mathsf{EnhanceGenReplicaAndMetadata}(\beta, K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s)$
   - 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 $\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}$ to each server
3. The client now deletes the file $\mathtt{F}$ and stores only a small, constant, amount of data: $K_1, \delta_1, \ldots, \delta_s$, and τ. K_2 and β are made public.

Challenge: Similar to the Challenge phase of RDC-SR.
Repair: Assume the client C has identified a faulty server with index y (*i.e.*, replica $\mathtt{F}_y$ 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 y.

1. Server S_y retrieves the replica $\mathtt{F}_i$ and the set of all verification tags from server S_i
2. S_y recovers the original file $\mathtt{F}$ by running $(\mathtt{F}) \leftarrow \mathsf{ButterflyDecode}(\beta, K_2, \mathtt{F}_i, i)$
3. S_y generates its own replica $\mathtt{F}_y$ by running $(\mathtt{F}_y) \leftarrow \mathsf{ButterflyEncode}(\beta, K_2, \mathtt{F}, y)$

Fig. 7. ERDC-SR: an Enhanced replication-based RDC system with Server-side Repair.

K_2. After having created t distinct replicas and the corresponding verification tags, the client outsources them to t different data centers (servers) of the CSP, so that each data center stores one replica and the set of all verification tags. The client then publishes K_1 and β. The Challenge phase of ERDC-SR is similar to that of RDC-SR. During the Repair phase, the client acts as the repair coordinator, coordinating the CSP's servers to repair the corrupted replicas.

We provide a detailed description of ERDC-SR in Figs 7 and 8, together with the following explanation of the three phases.

The Setup phase. The client generates two keys K_1 and K_2 by running KeyGen. It picks s random values $\delta_1, \ldots, \delta_s$ from GF(p), and determines threshold τ and dependency factor β according to the guidelines in Section 5.2. It then calls EnhanceGenReplicaAndMetadata t times to generate t distinct replicas and the corresponding verification tags. Each distinct replica will be stored in a different data center of the CSP, and the entire set of verification tags will be stored in each data center. The client then publishes key K_2 and β, deletes the original file and only keeps a small amount of data.

In EnhanceGenReplicaAndMetadata, the client calls ButterflyEncode to create a new replica, and computes a set of verification tags for this new replica. In ButterflyEncode, the client applies β-butterfly encoding over the collection of original file blocks (Fig. 6), in which the cryptographic transformation is based on a key derived from K_2 and the corresponding replica index.

The Repair phase. The client acts as the repair coordinator: It contacts the CSP, reports the corruption, and coordinates the CSP's servers to repair the corruption. A new server from the same data center is

```
EnhanceGenReplicaAndMetadata($\beta, K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s$):
1. Generate the $i$th replica: $(\mathtt{F}_i) \leftarrow$ ButterflyEncode($\beta, K_2, \mathtt{F}, i$)
2. Parse $\mathtt{F}_i$ as $\{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\}$
3. Compute verification tags:
   For $1 \leqslant j \leqslant n$: $\mathtt{z}_{ij} = h_{K_1}(i \parallel j) + \sum_{k=1}^{s} \delta_k \mathtt{m}_{ijk}$
4. Return $(\mathtt{z}_{i1}, \ldots, \mathtt{z}_{in}, \mathtt{F}_i = \{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\})$

ButterflyEncode($\beta, K_2, \mathtt{F}, i$):
1. Parse $\mathtt{F}$ as $\{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$
2. Initiate an array $G$ with $\mathtt{F}$, i.e., $G[0] = \mathtt{b}_1, G[1] = \mathtt{b}_2, \ldots, G[n-1] = \mathtt{b}_n$
3. Generate a key $K$ based on $K_2$ and $i$: $K = f_{K_2}(i)$
4. For $j$ from 1 to $\log \beta$ do
5. For $k$ from 0 to $\frac{n}{2^j} - 1$ do
6. For $l$ from 1 to $2^{j-1}$ do
7. $(G[l + k \cdot 2^j], G[l + k \cdot 2^j + 2^{j-1}]) \leftarrow E_K(G[l + k \cdot 2^j], G[l + k \cdot 2^j + 2^{j-1}])$
8. Return $(\mathtt{F}_i = \{\mathtt{m}_{i1} = G[0], \mathtt{m}_{i2} = G[1], \ldots, \mathtt{m}_{in} = G[n-1]\})$

ButterflyDecode($\beta, K_2, \mathtt{F}_i, i$):
1. Parse $\mathtt{F}_i$ as $\{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\}$
2. Initiate an array $G$ with $\mathtt{F}_i$, i.e., $G[0] = \mathtt{m}_{i1}, G[1] = \mathtt{m}_{i2}, \ldots, G[n-1] = \mathtt{b}_{in}$
3. Generate a key $K$ based on $K_2$ and $i$: $K = f_{K_2}(i)$
4. For $j$ from $\log \beta$ to 1 do
5. For $k$ from 0 to $\frac{n}{2^j} - 1$ do
6. For $l$ from 1 to $2^{j-1}$ do
7. $(G[l + k \cdot 2^j], G[l + k \cdot 2^j + 2^{j-1}]) \leftarrow D_K(G[l + k \cdot 2^j], G[l + k \cdot 2^j + 2^{j-1}])$
8. Return $(\mathtt{F} = \{\mathtt{b}_1 = G[0], \mathtt{b}_2 = G[1], \ldots, \mathtt{b}_n = G[n-1]\})$
```

Fig. 8. Components of ERDC-SR: EnhanceGenReplicaAndMetadata, ButterflyEncode, ButterflyDecode.

chosen to replace the corrupted server. The new server contacts one of the healthy servers, retrieves a replica, decodes it to restore the original file (ButterflyDecode), and encodes the original file to regenerate the corrupted replica (ButterflyEncode). The new server also retrieves the set of verification tags.

5.2. *Guidelines for using* ERDC-SR

In this section, we provide guidelines for using ERDC-SR in practice. We first provide an instantiation for the cryptographic transformation used in the β-butterfly encoding, and then provide guidelines on how to estimate various parameters used in the ERDC-SR scheme. For convenience, Fig. 9 provides a reference sheet for additional parameters used in ERDC-SR (refer to Fig. 5 for other parameters).

5.2.1. *Instantiating the cryptographic transformation*

The cryptographic transformation E used in ERDC-SR takes as input two RDC blocks and outputs two RDC blocks such that each bit of the output depends on each bit of the input, and both the output and the input are equal in size. Simply instantiating E with a block cipher such as AES is not enough, because there is a gap between the blocksize used by block ciphers and the blocksize used by RDC protocols (*e.g.*, PDP [4]): block ciphers like AES work with a small blocksize, *e.g.*, 128 bits for AES; however, the blocksize used in ERDC-SR as an RDC protocol should be large enough so that the storage overhead for the verification tags remains reasonable (*e.g.*, at least 4 KB). It may seem that this gap

- m: the number of 64-bit blocks in a RDC block.
- ρ: the annual growth rate of CSP's computational power.
- ϕ: the time period (in years, *e.g.*, 5 years), during which the client can obtain an assurance of the reliability and fault tolerance of the outsourced data, regardless of whether the CSP upgrades its computational power or not. After time period ϕ, the client should preprocess the data again.
- e: the time needed to compute one cryptographic transformation based on CSP's computational power at the time of Setup.
- u: the time needed to compute one AES operation over a 128-bit block based on CSP's computational power at the time of Setup.

Fig. 9. A reference sheet for all the parameters used in ERDC-SR.

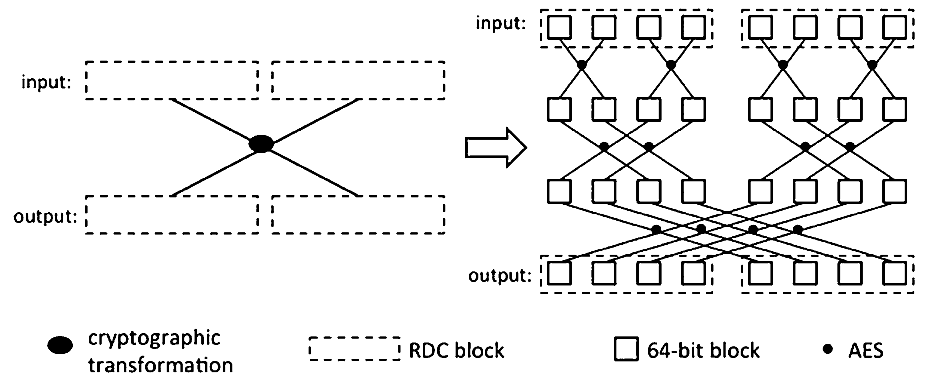


Fig. 10. An example for the instantiation of cryptographic transformation.

can be filled by using a block cipher with a mode of operation such as CBC [22]. However, this is not sufficient, since it does not necessarily guarantee the property that each bit of the output depends on each bit of the input. Instead, we construct E in two steps as follows:

(1) View the two input RDC blocks as a collection of $2 \cdot m$ 64-bit blocks.
(2) Apply a butterfly encoding (as shown in Fig. 6) over this collection of 64-bit blocks (at level 0) in a sequence of $\log(2 \cdot m)$ levels, such that the collection of blocks at level j is the output of a function (*i.e.*, AES over pairs of blocks) over level $j-1$, where $1 \leqslant j \leqslant \log(2 \cdot m)$.

This full butterfly encoding achieves "strong mixing" [41], such that each bit of the output (two RDC blocks) depends on each bit of the input (two RDC blocks). In Fig. 10, we show a concrete example for the instantiation of a cryptographic transformation where each RDC block has 4 64-bit blocks. Correspondingly, D (Section 5) can be instantiated as the reverse process of E. Note that m should be chosen as a power of 2, so that $\log(2 \cdot m)$ is always a positive integer.

5.2.2. *Estimating the parameters*

In the following, we first describe the best adversarial storage strategy for ERDC-SR, and then provide guidelines on how to estimate the scheme's parameters so that one can use it in practical applications. Among the parameters shown in Fig. 9, α, ρ and ϕ are known, and c can be determined according to prior work ([4]). We provide guidelines for estimating the remaining parameters τ, m, n, e and β.

The best storage strategy for the adversary in ERDC-SR. Similarly to RDC-SR, the best data distribution strategy for an α-cheating CSP is to store in each storage server $n\alpha$ blocks, which can be original file blocks, or intermediate blocks (Fig. 6), or blocks of the corresponding replica (*i.e.*, replica blocks), or a combination of these types of blocks. According to Appendix E, if the adversary only stores a block from level i ($0 \leqslant i \leqslant \log\beta$) of the butterfly network and is able to access the original file, then to generate a randomly challenged replica block, she needs to perform $\frac{\beta}{n \cdot 2^i} + \beta - \frac{\beta}{n} - 1$ cryptographic transformations. Let $f(i) = \frac{\beta}{n \cdot 2^i} + \beta - \frac{\beta}{n} - 1$. We observed that for fixed n and β, $f(i)$ decreases when i increases, *i.e.*, $f(i)$ is minimized when i is maximized. Thus, to minimize the effort of computing a proof to answer a challenge, the adversary should choose to store the blocks from the level corresponding to the maximum i in the butterfly network, which are the blocks from the corresponding replica.

Estimating τ. Similar to RDC-SR, the time threshold τ in ERDC-SR can be computed as $c \cdot x + 2 \cdot t_i$. Here x denotes the time each of the c challenged blocks contributes to the generation of the proof by the server at the time of Setup, 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.

Estimating m, n and e. Given $|F|$ (the size of a replica in bytes), n (the number of RDC blocks in a replica), and m (the number of 64-bit blocks in a RDC block), we have that $n \cdot m \cdot 8 = |F|$, or $n = \frac{|F|}{8 \cdot m}$. Since each replica should have at least 2 RDC blocks, *i.e.*, $n \geqslant 2$, we have $\frac{|F|}{8 \cdot m} \geqslant 2$. As described in Section 5.2.1, E is instantiated as a butterfly encoding with $\log(2 \cdot m)$ levels over 2 RDC blocks (*i.e.*, $2 \cdot m$ 64-bit blocks). Thus, we have

$$e = \frac{2 \cdot m}{2} \cdot \log(2 \cdot m) \cdot u = m \cdot \log(2 \cdot m) \cdot u \tag{3}$$

where u is the time needed to compute one AES operation over a 128-bit block during Setup. In Appendix F, we establish that $e > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{n+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$. Thus, we have $m \cdot \log(2 \cdot m) \cdot u > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{n+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$. Since $n = \frac{|F|}{8 \cdot m}$, we have $m \cdot \log(2 \cdot m) \cdot u > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{\frac{|F|}{8\cdot m}+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$, *i.e.*, $\log(2 \cdot m) \cdot u \cdot \frac{|F|}{8} + m \cdot \log(2 \cdot m) \cdot u \cdot (\frac{1}{2} \cdot c \cdot (1-\alpha) - 2) > ((1+\rho)^{\phi} - 1) \cdot \tau$, from which we can determine m by knowing $|F|$, c, α, u, ρ, ϕ and τ. Specifically, m should be chosen such that the following three conditions are satisfied:

- $\frac{|F|}{8 \cdot m} \geqslant 2$.
- $\log(2 \cdot m) \cdot u \cdot \frac{|F|}{8} + m \cdot \log(2 \cdot m) \cdot u \cdot (\frac{1}{2} \cdot c \cdot (1-\alpha) - 2) > ((1+\rho)^{\phi} - 1) \cdot \tau$.
- m is a power of 2.

We provide next an efficient algorithm (Algorithm 1) to find out the minimum value of m (*i.e.*, $m_{\min}$). Algorithm 1 keeps testing integers starting from 1. If it cannot find an integer that satisfies the aforementioned conditions, it will output -1, *i.e.*, such an $m_{\min}$ value does not exist. This may happen if $|F|$ is not large enough. For a concrete example, when $|F| = 100$ MB, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12$ sec, and $u = 0.1$ μs (estimated from our local machine, which is equipped with Intel Core *i5-4250U* processor and 4 GB RAM), $m_{\min}$ is 64. This gives the minimum size of an RDC block of 512 B.

After finding the minimum value of m, picking the exact m value is a trade-off between the storage overhead of verification tags and the computation/communication overhead during Challenge. Specifically, a larger m will lead to a larger RDC blocksize, which will lead to less storage overhead for verification tags, but more computation and communication overhead in each challenge. In practice, m is chosen such that the RDC blocksize is from KBs (*i.e.*, 4 KB) to tens of KBs (*i.e.*, 40 KB). In other words, $m_{\min}$ should be chosen small enough to allow flexibility on choosing a meaningful RDC blocksize.

We provide in Tables 2, 3, 4 and 5 the $m_{\min}$ value by varying the filesize under different sets of parameters. We observe that for a fixed set of parameters, in order to have a small $m_{\min}$, we should have a large enough filesize. For example, Table 2 shows that in order to choose m as small as 2048, the

Algorithm 1 Compute $m_{\min}$

```
Input: $|F|$, $c$, $\alpha$, $u$, $\rho$, $\phi$ and $\tau$;
Output: The minimum value of $m$;
1: $i = 0$;
2: $m = 1$;
3: while $\frac{|F|}{8 \cdot m} \geqslant 2$ AND $\log(2 \cdot m) \cdot u \cdot \frac{|F|}{8} + m \cdot \log(2 \cdot m) \cdot u \cdot (\frac{1}{2} \cdot c \cdot (1 - \alpha) - 2) \leqslant ((1 + \rho)^{\phi} - 1) \cdot \tau$ do
4:     $++i$;
5:     $m = 2^i$;
6: end while
7: if $\frac{|F|}{8 \cdot m} < 2$ then
8:     $m = -1$;
9: end if
10: return $m$
```

Table 2

$m_{\min}$ when $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12$ sec, and $u = 0.1$ μs

Filesize (MB)	50	60	70	80	90	100	120	140	180	230	350	500	1000
$m_{\min}$	8192	2048	512	256	128	64	32	16	8	4	2	2	1

Table 3

$m_{\min}$ when $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 5$, $\tau = 12$ sec, and $u = 0.1$ μs

Filesize (MB)	190	200	220	240	280	320	360	400	500	600	800	1000
$m_{\min}$	8192	4096	2048	1024	512	256	128	64	32	16	8	4

Table 4

$m_{\min}$ when $c = 460$, $\alpha = 0.9$, $\rho = 40\%$, $\phi = 2$, $\tau = 12$ sec, and $u = 0.1$ μs

Filesize (MB)	70	80	90	100	110	120	140	160	200	300	400	500
$m_{\min}$	8192	2048	1024	512	256	128	64	32	16	8	4	2

Table 5

$m_{\min}$ when $c = 460$, $\alpha = 0.9$, $\rho = 40\%$, $\phi = 5$, $\tau = 12$ sec, and $u = 0.1$ μs

Filesize (MB)	300	350	400	450	500	600	700	800	900	1000
$m_{\min}$	8192	2048	1024	512	256	128	64	32	16	16

filesize should be at least 60 MB. As another example, we can see in Table 3 that for a longer assurance interval ($\phi = 5$ years), the filesize should be at least 220 MB to allow for an m value as small as 2048.

By knowing $|F|$ and m, we can determine n, which is $\frac{|F|}{8 \cdot m}$. In the aforementioned example, we choose m as 512 (*i.e.*, the size of a RDC block is 4 KB), which is larger than the minimum value 64. Thus, n will be 25,600.

Once m is fixed, we can compute e based on Equation 3. Using the aforementioned example, when $m = 512$, we can compute $e = 512$ μs.

Estimating β. As mentioned earlier in this section, the α-cheating CSP should store in each storage server $n\alpha$ blocks of the corresponding replica in that server. Thus upon Challenge, when the client randomly checks c blocks in each replica, $c \cdot (1-\alpha)$ blocks will be missing on average. To answer the challenge from the client, the α-cheating server needs to generate these $c \cdot (1-\alpha)$ missing challenged blocks on the fly. By choosing β as large as n, we can always guarantee the α-cheating adversary cannot pass the verification by generating the missing challenged blocks on the fly. However, a large β will lead to an expensive Setup and Repair phase. Thus, we try to find a small β value after having fixed the parameters α, ρ, ϕ, c, τ, m, n, and e. This small β value should be chosen so that a dynamic α-cheating adversary cannot cheat successfully without being detected. Considering a certain time period ϕ over which the client wants to ensure the reliability and fault tolerance of the outsourced data, the CSP will always possess the most powerful computational capability at the end of ϕ, since it keeps growing its computational power over time. Thus, if we guarantee the adversary cannot cheat successfully at the end of ϕ, we will obtain a guarantee that it cannot cheat successfully at any time within ϕ. Upon answering a challenge issued by the client (in which a random subset of c replica blocks is challenged), a malicious server who is missing $n \cdot (1-\alpha)$ blocks, needs to first generate the $c \cdot (1-\alpha)$ missing blocks being challenged, and then compute a proof of data possession for this subset of c blocks. According to Appendix H, when the condition $n \geqslant 2 \cdot c \cdot (1-\alpha) \cdot \beta$ holds, the expected overall computation for creating the $c \cdot (1-\alpha)$ missing challenged blocks will be at least $\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta - 1)$ cryptographic transformations, where $p = \prod_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i}$. At the end of ϕ, the CSP's computational power will increase by $(1+\rho)^{\phi}$ times. Thus, generating the missing challenged blocks can be done in time at least $\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot \frac{e}{(1+\rho)^{\phi}}$, and computing the proof of data possession for the c challenged blocks can be done in time $\frac{cx}{(1+\rho)^{\phi}}$, which is approximately $\frac{\tau}{(1+\rho)^{\phi}}$. To ensure the malicious server cannot pass the verification at the end of ϕ, we have $\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot \frac{e}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}} > \tau$, *i.e.*, $\frac{\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot e}{(1+\rho)^{\phi}-1} > \tau$, from which we can estimate β by knowing n, c, α, e, ρ, ϕ and τ. Specifically, β should be chosen such that the following conditions can be satisfied simultaneously: First, $n \geqslant 2 \cdot c \cdot (1-\alpha) \cdot \beta$; Second, $\frac{\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot e}{(1+\rho)^{\phi}-1} > \tau$; Third, β is a power of 2. If we cannot find such a β value, we simply choose β as n. We provide next an efficient algorithm (Algorithm 2) which can find out the minimum β value. Algorithm 2 adopts a brute-force method: It keeps testing the integers starting from 1; after each run,

Algorithm 2 Estimate β

```
Input: $n, c, \alpha, e, \rho, \phi, \tau$;
Output: $\beta$;
 1: $j = 0$;
 2: $\beta = 1$;
 3: $p = \prod_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i}$;
 4: while $2 \cdot c \cdot (1-\alpha) \cdot \beta \leqslant n$ AND $\frac{\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot e}{(1+\rho)^{\phi}-1} \leqslant \tau$ do
 5:     $++j$;
 6:     $\beta = 2^j$;
 7:     $p = \prod_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i}$;
 8: end while
 9: if $2 \cdot c \cdot (1-\alpha) \cdot \beta > n$ then
10:     $\beta = n$;
11: end if
12: return $\beta$
```

the integer to be tested will be doubled. The computational complexity for Algorithm 2 is thus $O(\log n)$. The input parameters for Algorithm 2, n, c, α, e, ρ, ϕ and τ can be determined as follows:

(1) Pick α, ρ and ϕ, which are known in practical applications; pick c according to PDP [4].
(2) Follow the aforementioned guidelines to determine τ.
(3) Follow the aforementioned guidelines to determine the minimum value of m, and choose m such that it is larger than or equal to its minimum value.
(4) Compute n and e based on m.

Following the previous example, in which $|F| = 100$ MB, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12$ sec, $m = 512$, $n = 25{,}600$, and $e = 512$ μs, the output of Algorithm 2 will be 25,600, *i.e.*, each of the replica blocks should depend on all the original file blocks. As another example, when $|F| = 500$ MB, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12$ sec, $m = 512$ (by applying Algorithm 1, the minimum m will be 2 for this case, and we choose $m = 512$ to have a 4 KB RDC blocksize), $n = 128{,}000$, and $e = 512$ μs, the output of Algorithm 2 will be 1024, *i.e.*, each of the replica blocks should depend on 1024 original file blocks.

5.3. *Security analysis for* ERDC-SR

According to Section 5.2, ϕ is a time period after Setup, during which the client wants to obtain a guarantee that a dynamic α-cheating adversary cannot perform the ROTF attack without being detected. Let ϕ_{end} be an absolute point in time at the end of ϕ. At ϕ_{end}, the dynamic α-cheating adversary can simply be seen as a static α-cheating adversary that has the computational power corresponding to ϕ_{end}. Lemma 5.1 and Theorem 5.2 show that ERDC-SR can mitigate the ROTF attack executed by the adversary at ϕ_{end}. At any time before ϕ_{end}, the adversary has less computational power compared to ϕ_{end}. Thus, if it cannot successfully execute the ROTF attack without being detected at ϕ_{end}, it will not be able to do it at any time before ϕ_{end}. Thus, the client can obtain the aforementioned security guarantee at any time during ϕ before ϕ_{end}.

When ϕ expires, the client estimates a new set of τ and β parameters according to the guidelines in Section 5.2.2, and then retrieves the original file, pre-processes it again based on this new set of parameters, and outsources the replicas again. This will provide a similar security guarantee for the next time period. The client repeats this process until needed, thus obtaining a long-term security guarantee for its outsourced data.

Lemma 5.1. *In* ERDC-SR*, a cheating server who is storing an α-fraction of the corresponding replica, cannot generate at time ϕ_{end} the $(1-\alpha)c$ missing blocks and compute a proof to answer a challenge within time τ (here c is the number of file blocks checked by the client in a challenge, and the parameters τ and β are computed according to the guidelines in Section 5.2.2).*

Proof. According to Section 5.2.2, β is chosen so that the condition $P(E)\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot\frac{e}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}} > \tau$ always holds (recall $P(E)$ is the probability that all the $c\cdot(1-\alpha)$ missing challenged blocks depend on different sets of β original file blocks). To answer a challenge, a cheating server needs to first generate the $(1-\alpha)c$ missing blocks, and then compute a proof of data possession. Based on its computational power at ϕ_{end}, the cheating server needs at least $P(E)\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot\frac{e}{(1+\rho)^{\phi}}$ time for the first component, and at least $\frac{c\cdot x}{(1+\rho)^{\phi}}$ for the second component. Since τ is approximately $c\cdot x$, the cheating server needs at least $P(E)\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot\frac{e}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}}$, which is always greater than τ. $\square$

Theorem 5.2. *By choosing the β and τ parameters according to the guidelines in Section 5.2.2, the probability that an α-cheating adversary can successfully execute the ROTF attack at time ϕ_{end} without being detected is at most $\alpha^c c(1-\alpha)$ (here c is the number of file blocks checked by the client in a challenge).*

Proof. We have established in Section 5.2.2 that the best adversarial storage strategy for ERDC-SR is when each malicious server stores only an α fraction of the blocks of the corresponding replica. Thus each malicious server will be missing an $(1-\alpha)$ fraction of the replica blocks at ϕ_{end}.

As described in Section 5.2.2, the set of parameters τ and β in ERDC-SR is computed specifically for ϕ_{end}, based on the assumption that every time the client randomly checks c blocks from a replica stored in one of the t servers, at least $(1-\alpha)c$ blocks are from the missing $(1-\alpha)$ fraction of the replica, 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 a $(1-\alpha)$ fraction of the file blocks and the client randomly challenges c blocks, let V be the event that the cheating server is able to cheat successfully without being detected. In ERDC-SR, the client challenges all the storage servers simultaneously, and a cheating server cannot use other servers' computational power to compute a data possession proof for the challenged replica. Thus, event V happens when either (a) less than $(1-\alpha)c$ blocks are challenged among the file blocks that are missing at the server (event V_1), or (b) at least $(1-\alpha)c$ blocks are challenged among the missing file blocks but the cheating server is able to generate these missing challenged blocks and compute a proof to answer a challenge within time τ (event V_2). We compute the probability of V as $P(V) = P(V_1) + P(V_2)$.

According to Lemma 5.1, if we choose the parameters β and τ according to Section 5.2.2, the cheating server cannot generate the missing $(1-\alpha)c$ blocks and compute a proof within τ to pass a challenge successfully without being detected at ϕ_{end}. Thus, by choosing β and τ appropriately, we can ensure that event V_2 never happens, so $P(V_2) = 0$. From the proof of Theorem 4.2, $P(V_1) \leqslant \alpha^c c(1-\alpha)$, thus, $P(V) \leqslant \alpha^c c(1-\alpha)$. □

6. Implementation and evaluation

In this section, we first experimentally evaluate RDC-SR, and then evaluate the performance of ERDC-SR analytically. We also provide an evaluation on the storage cost of verification tags in both RDC-SR and ERDC-SR. Our experimental results show that RDC-SR can easily differentiate between benign and adversarial CSP behavior by using a time threshold; moreover, RDC-SR imposes a small load on the verifier in the Challenge phase and a small management load on the data owner in the Repair phase. Our analytical results show that ERDC-SR is very efficient in both Setup and Repair compared to SRDC-SR (a simple extension of RDC-SR to defend against a dynamic adversary, introduced at the beginning of Section 5).

6.1. *Experimental evaluation for* RDC-SR

6.1.1. *Background on Amazon's cloud services*

We first 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 8 and 9 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).

6.1.2. Experimental results

We build and test our prototype for RDC-SR on Amazon Web Services (AWS). Each server is run on an EC2 large instance (4 ECUs, 2 Cores, and 7.5 GB 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 3.0 GHz, with a 6144 KB cache), 333 GHz frontside bus, 4 GB RAM and a Hitachi HDP725032GLA360 360 GB 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 RDC-SR has been implemented in C and uses OpenSSL version 1.0.0e [29] for cryptographic operations. From Section 4.2, we have $\eta \geqslant \frac{x}{(1-\alpha)\cdot s\cdot t_{\text{prf}}}$ and we also choose $x = 0.034$ sec. We estimate $t_{\text{prf}} = 4.3$ 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 Section 4.2 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 do not 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 Section 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 [17,32]).

We measure the time for masking, verification tag generation and total preprocessing for one masked replica under three sets of (α, η) parameters. We repeat the experiments for four different file sizes (20 MB, 50 MB, 80 MB, and 100 MB). With blocks of 4 KB in size, the value of n (the number of file blocks) will be 5000, 12,500, 20,000, and 25,000, respectively. Table 6 shows the throughput for total preprocessing and its different components.

We have several observations for Table 6: 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 5 MB 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

Table 6

Preprocessing throughput

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

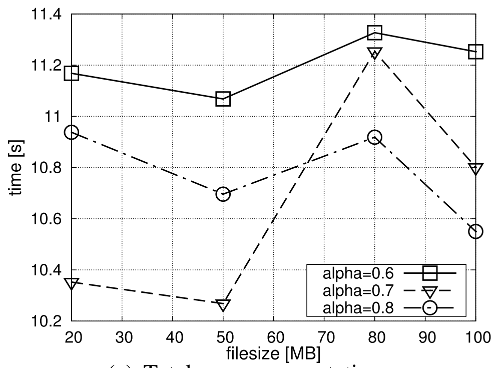
proof. For simplicity, we only challenge the server running in EC2 Oregon which is responsible for the replica stored in S3 Oregon. The number of blocks to be challenged is $c = 400$, which provides a high guarantee to detect data corruption by the server [4]. For the chosen values of α (Table 6) and c, the probability that a server performs the ROTF attack without being detected is less than $1.38 * 10^{-37}$ (cf. Section 4.3). 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, with each request being for a one-block range. This explains partially the large variation we observe in block access time for S3 (Figs 11(b) and 12(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 Section 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.

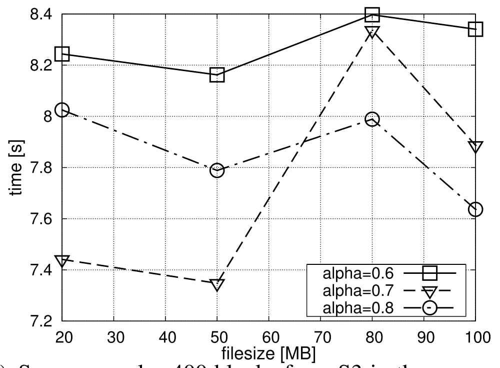
We repeat the experiments for different sets of (α, η) parameters and for different file sizes. Figure 11 and 12 show the server computation and client computation for both cases.

For the benign case, we observe from Fig. 11 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 [4] 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 11(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.

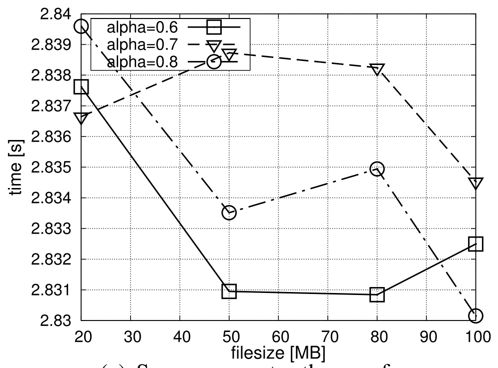
For the adversarial case, we observe from Fig. 12 that the total server computation and its various components are independent of filesize. The reason has been explained in the benign case. For Fig. 12(c),



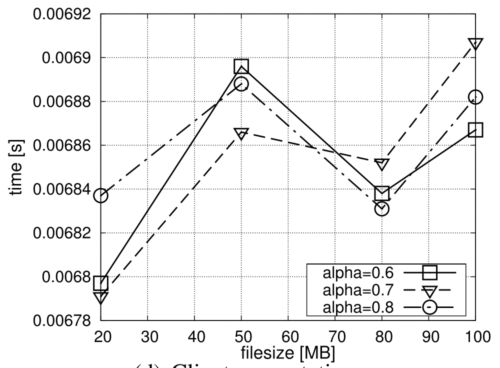
(a) Total server computation.



(b) Server samples 400 blocks from S3 in the same region.



(c) Server computes the proof.

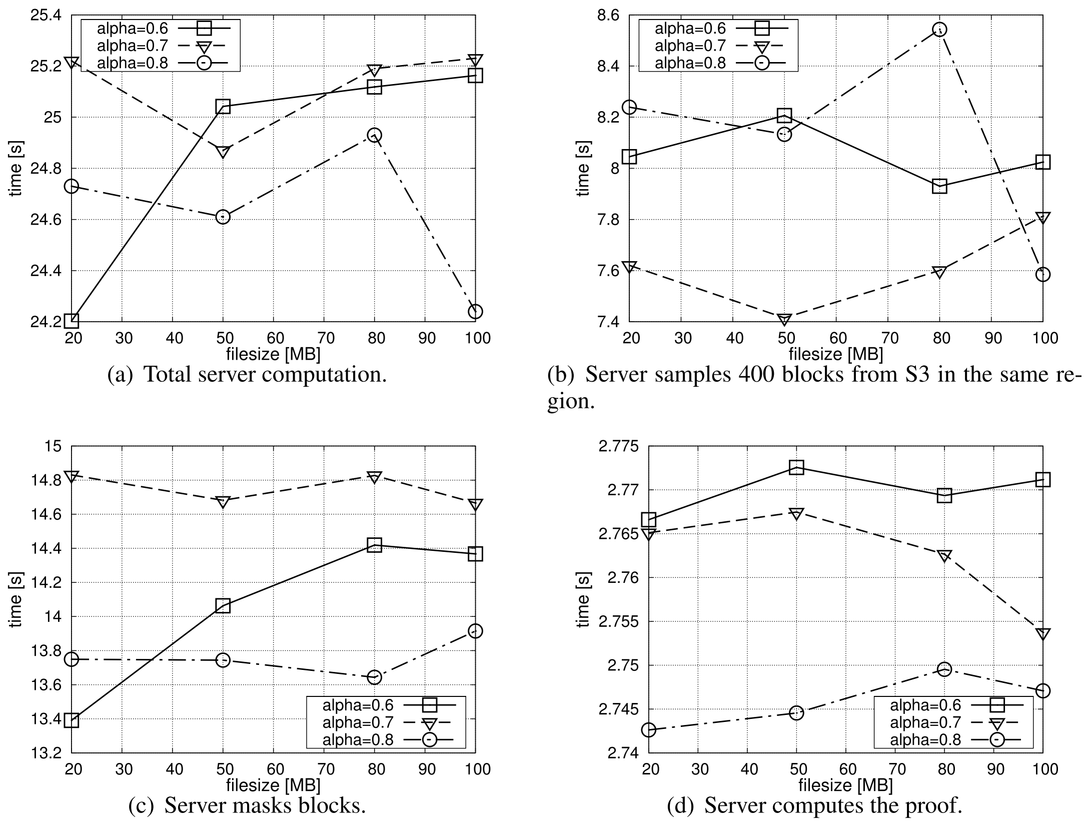


(d) Client computation.

Fig. 11. Computational cost for both the server and the client in challenge phase (benign case).

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, Fig. 12(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 $400(1-\alpha)\cdot s\cdot \eta\cdot t_{\text{prf}}$, 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 $400(1-\alpha)\cdot s\cdot\eta\cdot t_{\text{prf}} \geqslant 400(1-\alpha)\cdot s\cdot t_{\text{prf}}\cdot\frac{x}{(1-\alpha)\cdot s\cdot t_{\text{prf}}} = 400x = 13.6$ sec. We observe that most of the points in Fig. 12(c) are over this lower bound, except the point in 20 MB 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 Fig. 12(b) and Fig. 12(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 12(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%).



(a) Total server computation.

(b) Server samples 400 blocks from S3 in the same region.

(c) Server masks blocks.

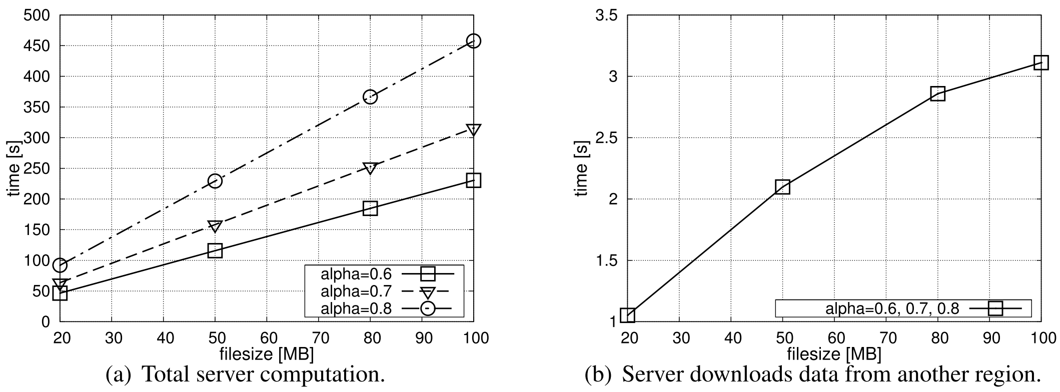
(d) Server computes the proof.

Fig. 12. Computational cost for the server and its various components in challenge phase (adversarial case).

According to the guidelines for establishing the time threshold τ in Section 4.2, τ should be 13.7 sec ($c = 400$, $x = 0.034$ sec, $t_i = 0.045$ sec based on our experiments). We see that 95% of the individual runs for Fig. 11(a) are below this threshold, and 100% of the individual runs for Fig. 12(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 Fig. 13 (this includes time for masking to generate a new replica, as described Table 6). We observe from Fig. 13(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.



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

Fig. 13. Computational cost for repairing a replica.

6.2. *Performance analysis for* ERDC-SR

In this section, we provide an analytical performance analysis for ERDC-SR by comparing ERDC-SR to SRDC-SR (recall that SRDC-SR, presented at the beginning of Section 5, is a simple extension of RDC-SR designed to handle dynamic α-cheating adversaries). In the following, we first estimate the parameters used in the analysis, and then evaluate the computational time needed to create a new replica from the original file during Setup for these two schemes. We then compare the computational time of these two schemes in both the Setup and the Repair phases. Let t_{prf} be the time required to compute one PRF during Setup in RDC-SR. Recall from Section 5.2 that e is defined as the time needed to compute one cryptographic transformation during Setup in ERDC-SR.

Estimating a concrete value for e. In the following, we estimate a concrete value for e which will be used in our analysis. According to Equation (3), e can be computed as $m \cdot \log(2 \cdot m) \cdot u$, where m is the number of 64-bit blocks in a RDC block and u is the time needed to compute one AES operation over a 128-bit block based on CSP's computational power during Setup (estimated as 0.1 μs in Section 5.2.2). Thus, we need to first estimate m and then estimate e. According to Section 5.2.2, after having computed $m_{\min}$, we choose m such that, (a) $m \geqslant m_{\min}$, and (b) the RDC blocksize is from KBs (*i.e.*, 4 KB) to tens of KBs (*i.e.*, 40 KB), and (c) m is a power of 2. In general, m can be a value from 512 to 4096. In the experimental evaluation of RDC-SR, we use 40 KB as the RDC blocksize. For consistency, we choose m as 4096 such that the RDC blocksize in ERDC-SR is close to 40 KB. Correspondingly, e is 5325 μs.

For RDC-SR, η is the masking factor which denotes the number of masking operations imposed on each symbol when generating a distinct replica (see Section 4). Thus, the time needed to create a new replica from the original file during Setup is approximately $n \cdot s \cdot \eta \cdot t_{\text{prf}}$ (the file is divided into n blocks, each of which consists of s symbols. When creating a new replica, each symbol is masked by adding η random values generated by the PRF). By estimating η as $\frac{x}{(1-\alpha)\cdot s\cdot t_{\text{prf}}}$ (Section 4.2), the time needed to create a new replica is $n \cdot s \cdot \eta \cdot t_{\text{prf}} = \frac{n\cdot x}{1-\alpha} = \frac{n\cdot c\cdot x}{c\cdot(1-\alpha)} \approx \frac{n\cdot\tau}{c\cdot(1-\alpha)}$.

To evaluate the time for creating a new replica in SRDC-SR from the original file during Setup, we first estimate the parameters τ and η used in SRDC-SR. For τ: similar to RDC-SR, τ is estimated as $c \cdot x + 2 \cdot t_i$. For η: upon answering a challenge issued by the client (in which a random subset of c replica blocks is challenged), a malicious server who is missing $n \cdot (1-\alpha)$ blocks, needs to first generate the $c \cdot (1-\alpha)$ missing blocks being challenged, and then compute a proof of data possession for this subset of c blocks. To generate the missing $c \cdot (1-\alpha)$ blocks, the computational time at the end of ϕ will be $c \cdot (1-\alpha) \cdot s \cdot \eta \cdot \frac{t_{\text{prf}}}{(1+\rho)^{\phi}}$ (generating one replica block requires $s \cdot \eta \cdot t_{\text{prf}}$ computational time during Setup). To compute the proof of data possession for the c challenged blocks, the computational time at the end of ϕ will be $\frac{cx}{(1+\rho)^{\phi}}$, which is approximately $\frac{\tau}{(1+\rho)^{\phi}}$. Thus, to ensure an α-cheating adversary cannot cheat successfully by performing the ROTF attack at the end of ϕ, we have: $c \cdot (1-\alpha) \cdot s \cdot \eta \cdot \frac{t_{\text{prf}}}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}} > \tau$, *i.e.*, $\eta > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot s\cdot t_{\text{prf}}}$. The time needed to create a new replica from the original file during Setup is $n \cdot s \cdot \eta \cdot t_{\text{prf}}$. By estimating η as $\frac{((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot s\cdot t_{\text{prf}}}$, the time needed to create a new replica becomes $\frac{n\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)}$. For ERDC-SR, the time required for creating a new replica from the original file during Setup is $\frac{n}{2} \cdot \log\beta \cdot e$ (to create a new replica in ERDC-SR, we apply a β-butterfly encoding, which has $\log\beta$ levels, and $\frac{n}{2}$ cryptographic transformations in each level). When estimating β, Algorithm 2 in Section 5.2.2 will return a β value for which we can distinguish two cases: $\beta = n$ and $\beta < n$.

- When $\beta = n$, the time for creating a distinct replica from the original file during Setup becomes $\frac{n}{2} \cdot \log n \cdot e$.
- When $\beta < n$, we approximate the β value as follows: Recall from Section 5.2.2 that $\frac{\frac{1+p}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1} > \tau$. To guarantee $\frac{\frac{1+p}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1}$ is always larger than τ, the lower bound of $\frac{\frac{1+p}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1}$ should be larger than τ. Since $p = \prod_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i} \geqslant 0$, the lower bound of $\frac{\frac{1+p}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1}$ is $\frac{\frac{1}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1}$. Thus, we have $\frac{\frac{1}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1} > \tau$, *i.e.*, $\beta > \frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e} + 1$. By estimating β as $\frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e} + 1$, the time for creating a distinct replica from the original file during Setup is $\frac{n}{2} \cdot \log(\frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e} + 1) \cdot e$.

Comparison between ERDC-SR and SRDC-SR. In the following, we compare ERDC-SR and SRDC-SR in both the Setup and the Repair phase. Based on what we have established previously, the overall computational time during Setup for ERDC-SR will be $t \cdot \frac{n}{2} \cdot \log n \cdot e$ when $\beta = n$, and $t \cdot \frac{n}{2} \cdot \log(\frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e} + 1) \cdot e$ when $\beta < n$. We use r to denote the ratio between the overall computational time of ERDC-SR and that of SRDC-SR during Setup.

- When $\beta = n$, r is $\frac{t\cdot\frac{n}{2}\cdot\log n\cdot e}{t\cdot\frac{n\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)}}$, which can be further reduced to $\frac{\log n\cdot e\cdot c\cdot(1-\alpha)}{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}$. To show some concrete values for r, we choose $c = 460$, $\alpha = 0.9$, $\tau = 12s$, $e = 5325\ \mu$s, $n = 25{,}600$, and vary ϕ and ρ (see Table 7). Note that in the aforementioned examples, each set of n, c, α, e, ρ, ϕ, τ values can guarantee β should be chosen as n (Algorithm 2 in Section 5.2.2). We observe from Table 7 that r is always smaller than 0.1, *i.e.*, for the case of $\beta = n$, the overall computational time of ERDC-SR in Setup is at least an order of magnitude less than that of SRDC-SR. Specifically for some case in Table 7 (*e.g.*, $\phi = 15$ and $\rho = 40\%$), the overall computational time of ERDC-SR in Setup can be 1000 times less expensive than that of SRDC-SR.
- When $\beta < n$, r is $\frac{t\cdot\frac{n}{2}\cdot\log(\frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e}+1)\cdot e}{t\cdot\frac{n\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)}}$, which can be further reduced to $\frac{t\cdot n\cdot\log\beta\cdot e}{t\cdot n\cdot(\beta-1)\cdot e}$, *i.e.*, $\frac{\log\beta}{\beta-1}$. We observe that r always decreases when β increases (note that β is a positive integer larger than 1). When $\beta = 64$, $r = \frac{\log 64}{64-1} = 0.095$. Considering e is in the range of milliseconds (*i.e.*, 10^{-3} s), β (*i.e.*, $\frac{2\cdot((1+\rho)^{\phi}-1)\cdot\tau}{c\cdot(1-\alpha)\cdot e} + 1$) should be on the order of 10^3, which is always larger than 64, *i.e.*, r is always smaller than 0.095. Thus, for the case of $\beta < n$, we conclude that the overall computational time of ERDC-SR in Setup is at least an order of magnitude less than that of SRDC-SR.

Table 7

Concrete values for r by varying ϕ and ρ when $\beta = n$ (recall that r is the ratio between the overall computational time of ERDC-SR and that of SRDC-SR)

	ρ	
ϕ	30%	40%
5	0.054	0.033
10	0.011	0.005
15	0.003	0.001

During Repair, the overall workload contains two components, decoding (unmasking) a replica to generate the original file and encoding (masking) the original file to generate the corresponding replica. Since decoding (unmasking) a replica is the reverse operation of encoding (masking) a replica, for both ERDC-SR and SRDC-SR, the overall computational time in the Repair phase is twice as much as that in the Setup phase, *i.e.*, the ratio of the overall computational time in Repair between ERDC-SR and SRDC-SR is the same as that in Setup. Thus, we conclude that the overall computational time of ERDC-SR in Repair is at least an order of magnitude less than that of SRDC-SR.

6.3. *Storage cost of the verification tags*

In both RDC-SR and ERDC-SR, we store in each storage server the entire set of verification tags (*i.e.*, the verification tags for all the t replicas). This will add a modest storage overhead to each server, since t has a small value in practical applications. For example, for 40 KB blocksize, 80 bit prime number p and $t = 50$, the additional storage in each server is only 1.25%. Even for large t, we can reduce the storage needed for these verification tags by encoding them with MDS codes (*e.g.*, erasure codes). We can use a (t, k) erasure code to distribute the entire set of verification tags to t servers, so that any k out of t servers have enough information to recover the entire set of verification tags. This approach will optimize the storage, but will require more computation because to repair the verification tags stored at a corrupted server, one will need to retrieve the encoded portions from k servers and decode them. The approach can be improved by using a $(t - 1, k)$ erasure code to distribute the entire set of verification tags to $t - 1$ servers (any k out of $t - 1$ servers have enough information to recover the entire set of verification tags), and storing in the remaining server the entire set of verification tags. During Repair, if the server that stores the entire set of verification tags is not corrupted, it can be directly used for repair purposes.

7. Related work

7.1. *Remote data checking*

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) [4] and Proofs of Retrievability (PoR) [27,34]. Later RDC schemes investigated models that can provide strong integrity guarantees while supporting dynamic operations on the outsourced data [6,11,12,23,36,39,40,43,48,49]. Recently, several RDC schemes have been designed to secure outsourced version control systems [14,24].

RDC for the multiple-server setting. RDC has been extended to the multiple-server setting (distributed RDC). Curtmola et al. proposed MR-PDP [16], 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. [8] and Wang et al. [42] built RDC schemes for erasure coding-based distributed storage systems. Chen et al. [15] proposed an RDC scheme for network coding-based distributed storage systems. All the aforementioned distributed RDC schemes 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 RDC schemes.

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. [9] propose RAFT, a new time-based RDC scheme that can be seen as a proof of fault tolerance; the scheme enables a client to obtain a proof that a given file is distributed across an expected number of physical storage devices in a single data center.

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 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 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, 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 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. [7] 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 paradigm.

Watson et al. [45] propose LoSt, which formalizes the concept of Proofs of Location (PoL). A PoL relies on a geolocation scheme [7] and a Proof of Retrievability (PoR) scheme. We summarize the differences between RDC-SR and LoSt. *First*, the goals are different. 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, 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. 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 there is no collusion between the CSP and the data centers. *Third*, the underlying idea for the solution is different. 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.

Gondree and Peterson [26] further relax the adversarial models and assumptions of previous PoL scheme [7], and propose a constraint-based data geolocation protocol that binds the latency-based geolocation techniques with PDP scheme.

Chen et al. [10] also explored the concept of server-side repair, but in the context of erasure code-based distributed storage systems that can function under an adversarial setting. Our focus in this work is on replication-based distributed storage systems.

7.2. Proofs of work (PoW)

CPU-bound PoW. Dwork and Naor [19] pioneered the concept of PoW, which was originally proposed to discourage junk emails. The underlying idea for a PoW is: when sending an email m, the sender is required to compute some moderately-hard function $f(m)$, and sends $(m, f(m))$ to the receiver; the receiver, however, can efficiently verify $f(m)$. Since sending an email requires to take a lot of additional work, the sender is discouraged from sending junk email. At the same time, the receiver will not be interrupted too much since verying $f(m)$ is efficient. They suggested some CPU-intensive candidates for function $f()$, *e.g.*, a function based on the signature scheme of Fiat and Shamir [25]. Both RDC-SR and ERDC-SR are CPU-bound PoWs, in which we either assume the CPU capability is bounded (RDC-SR) or assume the growth of the CPU capability is bounded (ERDC-SR).

Memory-bound PoW and proofs of space. Abadi et al. [1] observed sharp disparities of the CPU speed across distinct computer systems, *e.g.*, a PC always runs much faster than a PDA, and a high-end computer system which has sophisticated pipelines and other advantageous features runs much faster than a low-end machine. They thus investigated an alternative of PoW which measures the number of times the memory is accessed. Their approach was further improved in [18,20,44]. Recently, Dziembowski et al. [21] and Ateniese at al. [2,3] independently proposed a new concept of proofs of space, in which the prover needs to employ a specified amount of memory space in order to compute a proof of work. As it was pointed out in [3], the schemes proposed by Dziembowski et al. [21] are in effect a variant of memory-bound PoWs, since their prover can possibly trade off space with computation. Both RDC-SR and ERDC-SR schemes rely on CPU-bound rather than memory-bound PoW mechanisms because we do not have a significant disparity between different computer systems in our CSP-based setting. The sharp disparities among different computer systems observed in [1] are due to the heterogeneous nature of the applications like emails, *e.g.*, the email senders and the receivers can be all types of computing devices like servers, desktops, laptops, tablets, smart phones, etc. Our CSP setting, however, is homogeneous, *i.e.*, the computing devices in the data centers of the CSP are purely cloud servers.

Other work. Similar to the work of Reiter et al. [31], RDC-SR relies on the idea that only a prover which has the data can respond quickly enough to pass a challenge. Unlike RDC-SR however, their work is set in the context of P2P networks and the verifier (client) needs to keep the data for the verification purpose. Dijk et al. [41] proposed hourglass schemes, which can allow the data owner to obtain a guarantee that the outsourced files are encrypted at rest, when the cloud provider itself holds the corresponding encryption keys. They introduced an additional transformation, *i.e.*, an hourglass function, by which the encrypted file is transformed to an hourglass format, and the client challenges the provider on this new file format. A malicious cloud provider, who does not honestly store the expected hourglass file (encapsulating the encrypted file), will not be able to answer the challenge within a certain time threshold. They provided three constructions for the hourglass function based on a butterfly encoding, on a permutation that distributes uniformly its input symbols, and on the RSA signing function. In our ERDC-SR scheme, we adapt the butterfly encoding to create dependencies between replica blocks and the original file blocks.

8. Conclusion

In this paper, we propose two schemes for replication-based distributed storage systems, RDC-SR and ERDC-SR, both of which enable server-side repair and minimize the load on the client side. Compared to RDC-SR, ERDC-SR is more suitable for real-world settings, because it is designed to provide security guarantees against an untrusted CSP whose computational power can grow over time. We establish the effectiveness of these two schemes based on experiments using a prototype built on Amazon AWS (for RDC-SR) and on analytical performance analysis (for ERDC-SR).

The proposed schemes have several limitations. Our approach minimizes the load on the data owner during Repair by shifting the computational burden to the cloud servers. However, to overcome the "replicate on the fly" attack, we require that replica generation is time consuming, which in turn makes Repair time consuming. This limits the applicability of our schemes to scenarios in which Repair is a rare operation. Moreover, both of our schemes rely on a fixed time threshold to detect server misbehavior. While this approach proved to work well in our evaluation, we assumed an accurate estimation of the time needed to generate a proof by the server, a task that may not always be feasible. Finally, ERDC-SR can only be applied to files over a certain size.

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grants No. CNS 1054754, CNS 1409523, and DGE 1565478, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. A8650-15-C-7521. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, and AFRL. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

Appendix A. Measurements for the Amazon CSP

Tables 8 and 9 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 [47] to download a large file. To measure the propagation delay, we adopt the method introduced in [7] 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 8 and 9 are averaged over 20 runs.

Table 8

Download bandwidth (in MB/s)

	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 9

Propagation delay (in milliseconds). The standard deviation is included in parenthesis

	Virginia	N. California	Oregon
Virginia	0.58 (0.08)	40 (0.07)	44.1 (4.86)
N. California	40 (0.07)	0.761 (0.061)	11 (0.41)
Oregon	44.1 (4.86)	11 (0.41)	0.224 (0.056)
Our institution	5 (1.4)	40 (0.7)	45 (4.4)

Appendix B. Sampling blocks from Amazon S3

We wrote a program running in an EC2 instance (Amazon Virginia region) to randomly sample 4 KB blocks from S3 Virginia region. We collect the time in Table 10. All the results are averaged over 20 runs.

Table 10

The time for randomly sampling 4 KB blocks from S3 Virginia region

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

Appendix C. Proof of Theorem 3.2

Proof. (sketch) To prove this theorem, we first show that each of the t storage servers should store an equal fraction of the whole storage, and we then show that the blocks stored by each server should be the blocks from the corresponding replica for that server.

Let A denote the event that the α-cheating CSP remains undetected. Let A_i denote the event that server i can always pass the challenge successfully, where $1 \leqslant i \leqslant t$. P denotes the probability of a given event. Assume the client always checks a random subset of c blocks out of n file blocks during each challenge. We have, $P(A) = P(A_1 \cap A_2 \cap \cdots \cap A_t)$. In our setting, $A_1, A_2, \ldots, A_t$ can be seen as independent events, thus:

$$P(A) = P(A_1)P(A_2)\cdots P(A_t) \tag{4}$$

Let x_i denote the number of blocks actually stored in server i, and let n denote the number of blocks in a file copy, then $n - x_i$ blocks are missing in that server. We have:

$$\sum_{i=1}^{t}(x_i) = \alpha n t \tag{5}$$

Let τ (in seconds) be the time threshold, within which if the response from a server is not received, then that replica will be considered corrupted. Assume a server can generate λ blocks on the fly per second by performing the ROTF attack. (Note that this captures all types of ROTF attacks, which may have different values of λ.) According to prior work [4], we have:

$$P(A_i) = \frac{x_i + \lambda\tau}{n}\frac{x_i + \lambda\tau - 1}{n-1}\cdots\frac{x_i + \lambda\tau - c + 1}{n - c + 1} = \frac{\prod_{j=0}^{c-1}(x_i + \lambda\tau - j)}{\prod_{j=0}^{c-1}(n-j)} \tag{6}$$

According to Equations (4) and (6), we have:

$$\begin{aligned} P(A) &= \prod_{i=1}^{t}\left(\frac{\prod_{j=0}^{c-1}(x_i + \lambda\tau - j)}{\prod_{j=0}^{c-1}(n-j)}\right) \\ &= \frac{\prod_{i=1}^{t}(x_i + \lambda\tau)\prod_{i=1}^{t}(x_i + \lambda\tau - 1)\cdots\prod_{i=1}^{t}(x_i + \lambda\tau - c + 1)}{\prod_{i=1}^{t}\prod_{j=0}^{c-1}(n-j)}. \end{aligned}$$

Let function $f_j(x_1, \dots, x_t) = \prod_{i=1}^{t}(x_i + \lambda\tau - j)$, in which $0 \leqslant j \leqslant c-1$, then:

$$P(A) = \frac{f_0(x_1, \dots, x_t) f_1(x_1, \dots, x_t) \cdots f_{c-1}(x_1, \dots, x_t)}{\prod_{i=1}^{t}\prod_{j=0}^{c-1}(n-j)} \tag{7}$$

In practical applications, $x_i + \lambda\tau - j$ is always positive, in which $1 \leqslant i \leqslant t$ and $0 \leqslant j \leqslant c-1$. Thus, we have:

$$\text{For } 0 \leqslant j \leqslant c-1: \quad f_j(x_1, \dots, x_t) = \prod_{i=1}^{t}(x_i + \lambda\tau - j) \geqslant 0 \tag{8}$$

The best data distribution strategy for an α-cheating CSP should be a strategy that can maximize $P(A)$. According to Arithmetic Mean-Geometric Mean Inequality [38], we have: $(\prod_{i=1}^{t}(x_i + \lambda\tau))^{\frac{1}{t}} \leqslant \frac{\sum_{i=1}^{t}(x_i+\lambda\tau)}{t}$, and the equality holds if and only if $x_1 + \lambda\tau = x_2 + \lambda\tau = \cdots = x_t + \lambda\tau$, *i.e.*, if and only if $x_1 = x_2 = \cdots = x_t = n\alpha$ (Equation (5)). This is equivalent to $\prod_{i=1}^{t}(x_i + \lambda\tau) \leqslant (\frac{\sum_{i=1}^{t}(x_i+\lambda\tau)}{t})^t$, in which the equality holds if and only if $x_1 = x_2 = \cdots = x_t = n\alpha$. Similarly, we have the following inequality for the general case: $f_j(x_1, \dots, x_t) = \prod_{i=1}^{t}(x_i + \lambda\tau - j) \leqslant (\frac{\sum_{i=1}^{t}(x_i+\lambda\tau-j)}{t})^t$, where $0 \leqslant j \leqslant c-1$, and the equality holds if and only if $x_1 + \lambda\tau - j = x_2 + \lambda\tau - j = \cdots = x_t + \lambda\tau - j$, *i.e.*, the condition for maximizing $f_j(x_1, \dots, x_t)$ is:

$$\begin{aligned} &\text{For } 0 \leqslant j \leqslant c-1: \\ &\quad f_j(x_1, \dots, x_t) = \max\bigl(f_j(x_1, \dots, x_t)\bigr), \quad \text{if and only if} \quad x_1 = \cdots = x_t = n\alpha \end{aligned} \tag{9}$$

According to Equations (7), (8) and (9), and due to $\prod_{i=1}^{t}\prod_{j=0}^{c-1}(n-j) > 0$, $P(A) = \max(P(A))$ if and only if $x_1 = x_2 = \cdots = x_t = n\alpha$, *i.e.*, to maximize the probability that the α-cheating CSP remains undetected, we should store in each storage server an equal fraction of the whole storage. $\square$

Appendix D. Quantifying the computation required for generating one replica block from the original file blocks in ERDC-SR

From Fig. 6, we want to generate a block at level $\log\beta$ from the original file blocks which are at level 0. Each block at level $\log\beta$ depends on 2 blocks at level $\log\beta - 1$, *i.e.*, we need 1 cryptographic transformation in order to generate this block by knowing the 2 blocks at level $\log\beta - 1$. Similarly, each of the 2 blocks at level $\log\beta - 1$ depends on a different set of 2 blocks at level $\log\beta - 2$, *i.e.*, by knowing the 4 blocks at level $\log\beta - 2$, we need 2 cryptographic transformations in order to generate the 2 blocks at level $\log\beta - 1$. Similarly, we need 4 cryptographic transformations in order to generate the 4 blocks in level $\log\beta - 2$. We can simply infer that to generate the blocks at level 1, we need $2^{\log\beta-1}$ cryptographic transformations. Thus, the overall computation needed is $1+2+4+\cdots+2^{\log\beta-1} = \beta - 1$ cryptographic transformations.

Appendix E. Quantifying the computation needed to generate a replica block when the adversary only stores one intermediate block

In Fig. 6, there are $\log\beta + 1$ levels and n blocks at each level. Let i denote the level index, *i.e.*, $0 \leqslant i \leqslant \log\beta$. Assume the adversary only stores one block at this server, which can be either an original file block ($i = 0$), or an intermediate block ($0 < i < \log\beta$), or a block from the corresponding replica (*i.e.*, a replica block, $i = \log\beta$). We want to quantify the overall computation needed to generate a challenged replica block, considering the adversary only stores one block at level i and is able to access the original file (cf. Section 3.2.3).

Let $\mathcal{B}$ be the block at level i stored by the adversary. We categorize the replica blocks into two types: blocks which are related to $\mathcal{B}$ (type I), and blocks which are not related to $\mathcal{B}$ (type II). According to Fig. 6, there are $2^{\log\beta-i}$ type I blocks and $n - 2^{\log\beta-i}$ type II blocks. To compute a type II challenged replica block, the adversary needs to start from the original file blocks and perform $\beta - 1$ cryptographic transformations (cf. Appendix D). Otherwise, to compute a type I challenged replica block, the adversary does not need to compute the intermediate block $\mathcal{B}$ as well as all the other intermediate blocks needed to compute $\mathcal{B}$. Thus, the computation needed to generate such a replica block is $\beta - 1 - (2^i - 1) = \beta - 2^i$ cryptographic transformations. Because the replica blocks being challenged in the Challenge phase are picked randomly, each of them has an equal probability of being picked, *i.e.*, $\frac{1}{n}$. When picking a random replica block, the probability that it is a type I block is $\frac{2^{\log\beta-i}}{n}$, and the probability that it is a type II block is $\frac{n-2^{\log\beta-i}}{n}$. Thus, the expected overall computation needed to generate this block will be $\frac{2^{\log\beta-i}}{n}\cdot(\beta-2^i)+\frac{n-2^{\log\beta-i}}{n}\cdot(\beta-1)$ cryptographic transformations, which is $\frac{\beta}{n\cdot 2^i}+\beta-\frac{\beta}{n}-1$ cryptographic transformations.

Appendix F. Determining the minimum value of e

Intuitively, β and e together determine the amount of computational effort the α-cheating adversary needs to spend in order to cheat successfully without being detected. In other words, when β is larger, e can be smaller. However, β cannot exceed n. Thus, e should have a minimum value, *i.e.*, $e_{\min}$, by which even if β is as large as n (this can happen if the outsourced file is small in size), the α-cheating adversary cannot perform the ROTF attack without being detected. In the following, we try to determine $e_{\min}$. When β is equal to n, the computation needed to generate a missing replica block purely from the original file blocks will be $n - 1$ cryptographic transformations (cf. Appendix D). To generate all the $c \cdot (1 - \alpha)$ missing challenged blocks, the α-cheating server needs to perform at least $n + \frac{1}{2} \cdot c \cdot (1 - \alpha) - 2$ cryptographic transformations based on the following observations: the adversary needs $n - 1$ cryptographic transformations to generate the first missing block; to generate the remaining $c \cdot (1 - \alpha) - 1$ missing blocks, the adversary needs at least $\frac{c\cdot(1-\alpha)-2}{2}$ cryptographic transformations because, when finishing computing the first missing block, the adversary has already computed a lot of intermediate blocks, and can re-use these intermediate blocks to compute the remaining $c \cdot (1 - \alpha) - 1$ missing blocks by performing no less than $\frac{c\cdot(1-\alpha)-2}{2}$ cryptographic transformations. To ensure the malicious server cannot pass the verification at the end of ϕ, we have $(n + \frac{1}{2} \cdot c \cdot (1 - \alpha) - 2) \cdot \frac{e}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}} > \tau$, *i.e.*, $e > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{n+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$.

Appendix G. Quantifying the probability that all the $c \cdot (1 - \alpha)$ missing challenged blocks depend on different sets of β original file blocks

We can see from Fig. 6 that each of the replica blocks depends on a set of β original file blocks. In other words, a replica block with index i depends on a set of β original file blocks with indices in the range $[\lfloor \frac{i}{\beta} \rfloor \cdot \beta, (\lfloor \frac{i}{\beta} \rfloor + 1) \cdot \beta - 1]$, where $0 \leqslant i \leqslant n - 1$. During Challenge, the client checks a random subset of c replica blocks. To pass the verification check, the α-cheating server needs to generate the $c \cdot (1 - \alpha)$ missing challenged blocks on the fly. Let E be the event that all the $c \cdot (1 - \alpha)$ missing challenged blocks depend on different sets of β original file blocks. We evaluate next $P(E)$, the probability of E.

We see from Fig. 6 that all the replica blocks with indices in the range $[\lfloor \frac{i}{\beta} \rfloor \cdot \beta, (\lfloor \frac{i}{\beta} \rfloor + 1) \cdot \beta - 1]$ depend on the same set of β original file blocks, where $0 \leqslant i \leqslant n - 1$. We define a dependency group as the collection of replica blocks from this same range. This gives a total of $\frac{n}{\beta}$ dependency groups. Thus, evaluating $P(E)$ is equivalent to evaluating the probability that all the $c \cdot (1 - \alpha)$ missing challenged blocks are from different dependency groups. Let E_j be the event that the jth missing challenged block $(1 \leqslant j \leqslant c \cdot (1 - \alpha))$ is from a different dependency group than the previously missing challenged blocks. Thus, $P(E) = \prod_{j=1}^{c\cdot(1-\alpha)} P(E_j)$. We now evaluate $P(E_j)$:

$$P(E_1) = 1$$

$$P(E_2) = \frac{\beta(\frac{n}{\beta} - 1)}{n - 1} = \frac{n - \beta}{n - 1}$$

$$P(E_3) = \frac{\beta(\frac{n}{\beta} - 2)}{n - 2} = \frac{n - 2\beta}{n - 2}$$

...

$$P(E_{c\cdot(1-\alpha)}) = \frac{\beta(\frac{n}{\beta} - (c\cdot(1-\alpha)-1))}{n-(c\cdot(1-\alpha)-1)} = \frac{n-(c\cdot(1-\alpha)-1)\beta}{n-(c\cdot(1-\alpha)-1)}$$

Thus, $P(E) = \prod_{j=1}^{c\cdot(1-\alpha)-1} \frac{n-j\beta}{n-j}$

Appendix H. Quantifying the minimum computation for an α-cheating server to generate the $c\cdot(1-\alpha)$ missing challenged blocks

During Challenge, the client checks a random subset of c replica blocks. To pass the check successfully without being detected, an α-cheating server needs to generate the $c\cdot(1-\alpha)$ missing challenged blocks on the fly. By applying the β-butterfly encoding, each of the replica blocks depends on a set of β original file blocks. Specifically, as shown in Fig. 6, a replica block with index i depends on β original file blocks with indices in the range $[\lfloor\frac{i}{\beta}\rfloor\cdot\beta, (\lfloor\frac{i}{\beta}\rfloor+1)\cdot\beta-1]$, where $0\leqslant i\leqslant n-1$. Let E_x be the event that these $c\cdot(1-\alpha)$ missing blocks depend on x different sets of β original file blocks, where $1\leqslant x\leqslant c\cdot(1-\alpha)$. When E_x happens, the overall computation required to generate the $c\cdot(1-\alpha)$ missing blocks will be at least $x\cdot(\beta-1)$ cryptographic transformation because, among these $c\cdot(1-\alpha)$ missing blocks, the adversary needs to generate at least x blocks purely from the original file (*i.e.*, cannot re-use any intermediate blocks), and generating a block purely from the original file requires $\beta-1$ cryptographic transformation (cf. Appendix D). Let $P(E_x)$ be the probability of E_x, thus, the expected overall computation required by the adversary to generate these $c\cdot(1-\alpha)$ missing challenged blocks should be at least $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1)$ cryptographic transformations.

Computing all the $P(E_x)$ values, for $1\leqslant x\leqslant c\cdot(1-\alpha)$, will be quite complex. Thus, rather than computing the exact value for $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1)$, we evaluate its lower bound. We observe that when $n\geqslant 2\cdot c\cdot(1-\alpha)\cdot\beta$, we have $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1}) > \cdots > P(E_1)$. This is based on the following analysis:

- For the base case, we show that $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1})$. From Appendix G, we have $P(E_{c\cdot(1-\alpha)}) = \prod_{j=1}^{c\cdot(1-\alpha)-1}\frac{n-j\beta}{n-j}$. Similarly, we can compute $P(E_{c\cdot(1-\alpha)-1}) = (\prod_{j=1}^{c\cdot(1-\alpha)-2}\frac{n-j\beta}{n-j})\cdot\frac{(c\cdot(1-\alpha)-1)\cdot(\beta-1)}{n-(c\cdot(1-\alpha)-1)}$. When $n\geqslant 2\cdot c\cdot(1-\alpha)\cdot\beta$, then $n-(c\cdot(1-\alpha)-1)\cdot\beta > c\cdot(1-\alpha)\cdot\beta > 0$. Since $0 < (c\cdot(1-\alpha)-1)\cdot\beta < c\cdot(1-\alpha)\cdot\beta$, we get that $\frac{P(E_{c\cdot(1-\alpha)})}{P(E_{c\cdot(1-\alpha)-1})} = \frac{n-(c\cdot(1-\alpha)-1)\cdot\beta}{(c\cdot(1-\alpha)-1)(\beta-1)} > 1$, *i.e.*, $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1})$.
- For the general case, we show that $P(E_{x+1}) > P(E_x)$, where $1\leqslant x\leqslant c\cdot(1-\alpha)-1$. For event E_x, when picking the $c\cdot(1-\alpha)$ missing blocks, we first pick x blocks from all the $\frac{n}{\beta}$ dependency groups (Event $E_{x,1}$), such that each block is from a different dependency group, *i.e.*, x dependency groups have been picked; we then pick the remaining $c\cdot(1-\alpha)-x$ blocks from these x dependency groups (Event $E_{x,2}$). For event E_{x+1}, similarly, we first pick x blocks among all the $\frac{n}{\beta}$ dependency groups (Event $E_{x+1,1}$), such that each block is from a different dependency group; we then pick the remaining $c\cdot(1-\alpha)-x$ blocks in two steps (Event $E_{x+1,2}$): step 1, pick 1 block from the other $\frac{n}{\beta}-x$ dependency groups, *i.e.*, after step 1, $x+1$ dependency groups have been picked; step 2, pick the $c\cdot(1-\alpha)-x-1$ blocks from these $x+1$ dependency groups. $E_{x,1}$ and $E_{x,2}$ are two independent events, and E_x happens when both $E_{x,1}$ and $E_{x,2}$ happen, thus, we have $P(E_x) = P(E_{x,1})\cdot P(E_{x,2})$.

Similarly, we have $P(E_{x+1}) = P(E_{x+1,1}) \cdot P(E_{x+1,2})$. Since $n \geqslant 2 \cdot c \cdot (1-\alpha) \cdot \beta$, we have $\frac{n}{\beta} > 2 \cdot c \cdot (1-\alpha)$. Since $x \leqslant c \cdot (1-\alpha) - 1$ and $\frac{n}{\beta} > 2 \cdot c \cdot (1-\alpha)$, we have $\frac{n}{\beta} - x > c \cdot (1-\alpha)$, *i.e.*, $\frac{n}{\beta} - x > x$. Since $\frac{n}{\beta} - x > x$ and $x + 1 > x$, we conclude that $P(E_{x+1,2}) > P(E_{x,2})$. Since $P(E_{x+1,1}) = P(E_{x,1})$, we have $P(E_{x+1}) > P(E_x)$.

Based on the previous observation, we state Theorem H.1.

Theorem H.1. *When $n \geqslant 2 \cdot c \cdot (1-\alpha) \cdot \beta$, we have $\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x) \cdot x > \frac{1}{2} \cdot (1 - P(E_{c\cdot(1-\alpha)})) \cdot c \cdot (1-\alpha)$.*

Proof. We first show the following inequality:

$$P(E_x) \cdot x + P(E_{c\cdot(1-\alpha)-x}) \cdot \big(c \cdot (1-\alpha) - x\big) > \frac{1}{2}\big(P(E_x) + P(E_{c\cdot(1-\alpha)-x})\big) \cdot \big(c \cdot (1-\alpha)\big) \qquad (10)$$

always holds when $\frac{1}{2} \cdot c \cdot (1-\alpha) < x < c \cdot (1-\alpha)$ and $n \geqslant 2 \cdot c \cdot (1-\alpha) \cdot \beta$.

We justify inequality (10) by showing that $P(E_x) \cdot x + P(E_{c\cdot(1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) - \frac{1}{2}(P(E_x) + P(E_{c\cdot(1-\alpha)-x})) \cdot (c \cdot (1-\alpha)) > 0$. We have that:

$$\begin{aligned}
&P(E_x) \cdot x + P(E_{c\cdot(1-\alpha)-x}) \cdot \big(c \cdot (1-\alpha) - x\big) - \frac{1}{2}\big(P(E_x) + P(E_{c\cdot(1-\alpha)-x})\big) \cdot \big(c \cdot (1-\alpha)\big) \\
&\quad = P(E_x) \cdot \big(c \cdot (1-\alpha) - \big(c \cdot (1-\alpha) - x\big)\big) + P(E_{c\cdot(1-\alpha)-x}) \cdot \big(c \cdot (1-\alpha) - x\big) \\
&\qquad - \frac{1}{2}\big(P(E_x) + P(E_{c\cdot(1-\alpha)-x})\big) \cdot \big(c \cdot (1-\alpha)\big) \\
&\quad = \frac{1}{2} \cdot P(E_x) \cdot c \cdot (1-\alpha) - \frac{1}{2} \cdot P(E_{c\cdot(1-\alpha)-x}) \cdot c \cdot (1-\alpha) \\
&\qquad - \big(P(E_x) - P(E_{c\cdot(1-\alpha)-x})\big) \cdot \big(c \cdot (1-\alpha) - x\big) \\
&\quad = \big(P(E_x) - P(E_{c\cdot(1-\alpha)-x})\big)\left(x - \frac{1}{2} \cdot c \cdot (1-\alpha)\right).
\end{aligned}$$

Since $x > \frac{1}{2} \cdot c \cdot (1-\alpha)$, we have $x - \frac{1}{2} \cdot c \cdot (1-\alpha) > 0$. Since $\frac{1}{2} \cdot c \cdot (1-\alpha) < x < c \cdot (1-\alpha)$, we have $c \cdot (1-\alpha) - x < x$, thus, $P(E_{c\cdot(1-\alpha)-x}) < P(E_x)$, *i.e.*, $P(E_x) - P(E_{c\cdot(1-\alpha)-x}) > 0$. We can further infer that $(P(E_x) - P(E_{c\cdot(1-\alpha)-x}))(x - \frac{1}{2} \cdot c \cdot (1-\alpha)) > 0$, or equivalently, $P(E_x) \cdot x + P(E_{c\cdot(1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) - \frac{1}{2}(P(E_x) + P(E_{c\cdot(1-\alpha)-x})) \cdot (c \cdot (1-\alpha)) > 0$.

Based on inequality (10), we prove the theorem by differentiating between two cases: when $c \cdot (1-\alpha)$ is odd and when $c \cdot (1-\alpha)$ is even.

1) When $c \cdot (1-\alpha)$ is odd:

$$\begin{aligned}
&\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x) \cdot x \\
&\quad = P(E_{c\cdot(1-\alpha)-1}) \cdot \big(c \cdot (1-\alpha) - 1\big) + P(E_1) \cdot 1 + P(E_{c\cdot(1-\alpha)-2}) \cdot \big(c \cdot (1-\alpha) - 2\big)
\end{aligned}$$

$$\begin{aligned}
&+ P(E_2)\cdot 2 + \cdots + P(E_{\lceil \frac{1}{2}\cdot c\cdot(1-\alpha)\rceil})\cdot\left(\left\lceil \frac{1}{2}\cdot c\cdot(1-\alpha)\right\rceil\right)\\
&+ P(E_{\lfloor \frac{1}{2}\cdot c\cdot(1-\alpha)\rfloor})\cdot\left(\left\lfloor \frac{1}{2}\cdot c\cdot(1-\alpha)\right\rfloor\right)\\
&> \frac{1}{2}\big(P(E_{c\cdot(1-\alpha)-1}) + P(E_1)\big)\cdot\big(c\cdot(1-\alpha)\big) + \frac{1}{2}\big(P(E_{c\cdot(1-\alpha)-2}) + P(E_2)\big)\cdot\big(c\cdot(1-\alpha)\big)\\
&+ \cdots + \frac{1}{2}\big(P(E_{\lceil \frac{1}{2}\cdot c\cdot(1-\alpha)\rceil}) + P(E_{\lceil \frac{1}{2}\cdot c\cdot(1-\alpha)\rceil})\big)\cdot\big(c\cdot(1-\alpha)\big)\\
&= \frac{1}{2}\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\big(c\cdot(1-\alpha)\big)\\
&= \frac{1}{2}\cdot\big(1 - P(E_{c\cdot(1-\alpha)})\big)\big(c\cdot(1-\alpha)\big).
\end{aligned}$$

2) When $c\cdot(1-\alpha)$ is even:

$$\begin{aligned}
&\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x\\
&= P(E_{c\cdot(1-\alpha)-1})\cdot\big(c\cdot(1-\alpha)-1\big) + P(E_1)\cdot 1 + P(E_{c\cdot(1-\alpha)-2})\cdot\big(c\cdot(1-\alpha)-2\big)\\
&+ P(E_2)\cdot 2 + \cdots + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)+1})\cdot\left(\frac{1}{2}\cdot c\cdot(1-\alpha)+1\right)\\
&+ P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)-1})\cdot\left(\frac{1}{2}\cdot c\cdot(1-\alpha)-1\right) + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)})\cdot\frac{1}{2}\cdot c\cdot(1-\alpha)\\
&> \frac{1}{2}\big(P(E_{c\cdot(1-\alpha)-1}) + P(E_1)\big)\cdot\big(c\cdot(1-\alpha)\big) + \frac{1}{2}\big(P(E_{c\cdot(1-\alpha)-2}) + P(E_2)\big)\cdot\big(c\cdot(1-\alpha)\big)\\
&+ \cdots + \frac{1}{2}\big(P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)+1}) + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)-1})\big)\cdot\big(c\cdot(1-\alpha)\big)\\
&+ \frac{1}{2}\cdot P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha)\\
&= \frac{1}{2}\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\big(c\cdot(1-\alpha)\big)\\
&= \frac{1}{2}\cdot\big(1 - P(E_{c\cdot(1-\alpha)})\big)\big(c\cdot(1-\alpha)\big). \qquad \square
\end{aligned}$$

We are now ready to evaluate the lower bound of $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1)$, which is the expected overall computation required by the adversary to generate the $c\cdot(1-\alpha)$ missing challenged blocks.

We have:

$$\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1)$$

$$= (\beta-1)\cdot\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x$$

$$= (\beta-1)\cdot\left(\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x + P(E_{c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha)\right).$$

Based on Theorem H.1, when $n \geqslant 2\cdot c\cdot(1-\alpha)\cdot\beta$, we have $\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x > \frac{1}{2}\cdot(1-P(E_{c\cdot(1-\alpha)}))c\cdot(1-\alpha)$. Thus, $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1) > (\beta-1)\cdot(\frac{1}{2}\cdot(1-P(E_{c\cdot(1-\alpha)}))\cdot c\cdot(1-\alpha)+P(E_{c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha))$. Finally, this means that:

$$\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1)$$

$$> \frac{1}{2}\cdot\bigl(1+P(E_{c\cdot(1-\alpha)})\bigr)\cdot c\cdot(1-\alpha)\cdot(\beta-1)$$

$$= \frac{1+\prod_{i=1}^{c\cdot(1-\alpha)-1}\frac{n-i\beta}{n-i}}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1),\quad \text{when } n \geqslant 2\cdot c\cdot(1-\alpha)\cdot\beta.$$

References

[1] M. Abadi, M. Burrows, M. Manasse and T. Wobber, Moderately hard, memory-bound functions, *ACM Transactions on Internet Technology (TOIT)* **5**(2) (2005), 299–327. doi:10.1145/1064340.1064341.

[2] G. Ateniese, I. Bonacina, A. Faonio and N. Galesi, Proofs of space: When space is of the essence, in: *Proc. of the 9th International Conference, Security and Cryptography for Networks (SCN 2014)*, Springer International Publishing, 2014, pp. 538–557.

[3] G. Ateniese, I. Bonacina, A. Faonio and N. Galesi, Proofs of space: When space is of the essence, http://eprint.iacr.org/2013/805.pdf.

[4] 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** (2011).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] B. Chen, A.K. Ammula and R. Curtmola, Towards server-side repair for erasure coding-based distributed storage systems, in: *Proc. of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*, 2015.

[11] 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.
[12] 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.
[13] B. Chen and R. Curtmola, Towards self-repairing replication-based storage systems using untrusted clouds, in: *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013.
[14] B. Chen and R. Curtmola, Auditable version control systems, in: *Proc. of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.
[15] B. Chen, R. Curtmola, G. Ateniese and R. Burns, Remote data checking for network coding-based distributed storage systems, in: *Proc. of ACM Cloud Computing Security Workshop (CCSW'10)*, 2010.
[16] R. Curtmola, O. Khan, R. Burns and G. Ateniese, MR-PDP: Multiple-replica provable data possession, in: *Proc. of International Conference on Distributed Computing Systems (ICDCS'08)*, 2008.
[17] A.G. Dimakis, P.B. Godfrey, Y. Wu, M.O. Wainwright and K. Ramchandran, Network coding for distributed storage systems, *IEEE Trans. on Inf. Theory* **56** (2010).
[18] C. Dwork, A. Goldberg and M. Naor, On memory-bound functions for fighting spam, in: *Advances in Cryptology (Crypto'03)*, 2003.
[19] C. Dwork and M. Naor, Pricing via processing or combatting junk mail, in: *Advances in Cryptology (CRYPTO'92)*, 1993.
[20] C. Dwork, M. Naor and H. Wee, Pebbling and proofs of work, in: *Advances in Cryptology (CRYPTO'05)*, 2005.
[21] S. Dziembowski, S. Faust, V. Kolmogorov and K. Pietrzak, Proofs of space, http://eprint.iacr.org/2013/796.pdf.
[22] W.F. Ehrsam, C.H. Meyer, J.L. Smith and W.L. Tuchman, Message verification and transmission error detection by block chaining, Patent 4,074,066, Google Patents, 1978.
[23] 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.
[24] M. Etemad and A. Kupcu, Transparent, distributed, and replicated dynamic provable data possession, in: *Proc. of 11th International Conference on Applied Cryptography and Network Security (ACNS'13)*, 2013.
[25] A. Fiat and A. Shamir, How to prove yourself: Practical solutions to identification and signature problems, in: *Advances in Cryptology (CRYPTO'86)*, 1986.
[26] M. Gondree and Z.N.J. Peterson, Geolocation of data in the cloud, in: *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013.
[27] 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.
[28] H. Krawczyk, LFSR-based hashing and authentication, in: *Proc. of Annual International Cryptology Conference (CRYPTO'94)*, 1994.
[29] OpenSSL, http://www.openssl.org/.
[30] 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.
[31] M.K. Reiter, V. Sekar, C. Spensky and Z. Zhang, Making peer-assisted content distribution robust to collusion using bandwidth puzzles, *Information Systems Security* (2009), 132–147.
[32] 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.
[33] P. Rogaway, Bucket hashing and its application to fast message authentication, in: *Proc. of Annual International Cryptology Conference (CRYPTO'95)*, 1995.
[34] 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.
[35] H. Shacham and B. Waters, Compact proofs of retrievability, *J. Cryptology* **26**(3) (2013), 442–483. doi:10.1007/s00145-012-9129-2.
[36] E. Shi, E. Stefanov and C. Papamanthou, Practical dynamic proofs of retrievability, in: *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, 2013.
[37] V. Shoup, On fast and provably secure message authentication based on universal hashing, in: *Proc. of Annual International Cryptology Conference (CRYPTO'96)*, 1996.
[38] J.M. Steele, *The Cauchy–Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*, Cambridge University Press, 2004.
[39] 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.
[40] S.R. Tate, R. Vishwanathan and L. Everhart, Multi-user dynamic proofs of data possession using trusted hardware, in: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013.

[41] M. van Dijk, A. Juels, A. Oprea, R.L. Rivest, E. Stefanov and N. Triandopoulos, Hourglass schemes: How to prove that cloud files are encrypted, in: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012.
[42] 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.
[43] 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) (2011). doi:10.1109/TPDS.2010.183.
[44] B. Waters, A. Juels, J.A. Halderman and E.W. Felten, New client puzzle outsourcing techniques for DoS resistance, in: *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, 2004.
[45] 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.
[46] 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) (1981), 265–279. doi:10.1016/0022-0000(81)90033-7.
[47] Wget, http://www.gnu.org/software/wget/.
[48] Y. Zhang and M. Blanton, Efficient dynamic provable possession of remote data via balanced update trees, in: *Proc. of 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13)*, 2013.
[49] 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.