Summer 8-31-2014

New directions for remote data integrity checking of cloud storage

Bo Chen
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

NEW DIRECTIONS FOR REMOTE DATA INTEGRITY CHECKING OF CLOUD STORAGE

by
Bo Chen

Cloud storage services allow data owners to outsource their data, and thus reduce their workload and cost in data storage and management. However, most data owners today are still reluctant to outsource their data to the cloud storage providers (CSP), simply because they do not trust the CSPs, and have no confidence that the CSPs will secure their valuable data. This dissertation focuses on Remote Data Checking (RDC), a collection of protocols which can allow a client (data owner) to check the integrity of data outsourced at an untrusted server, and thus to audit whether the server fulfills its contractual obligations.

Robustness has not been considered for the dynamic RDCs in the literature. The R-DPDP scheme being designed is the first RDC scheme that provides robustness and, at the same time, supports dynamic data updates, while requiring small, constant, client storage. The main challenge that has to be overcome is to reduce the client-server communication during updates under an adversarial setting. A security analysis for R-DPDP is provided.

Single-server RDCs are useful to detect server misbehavior, but do not have provisions to recover damaged data. Thus in practice, they should be extended to a distributed setting, in which the data is stored redundantly at multiple servers. The client can use RDC to check each server and, upon having detected a corrupted server, it can repair this server by retrieving data from healthy servers, so that the reliability level can be maintained. Previously, RDC has been investigated for replication-based and erasure

coding-based distributed storage systems. However, RDC has not been investigated for network coding-based distributed storage systems that rely on untrusted servers. RDC-NC is the first RDC scheme for network coding-based distributed storage systems to ensure data remain intact when faced with data corruption, replay, and pollution attacks. Experimental evaluation shows that RDC-NC is inexpensive for both the clients and the servers.

The setting considered so far outsources the storage of the data, but the data owner is still heavily involved in the data management process (especially during the repair of damaged data). A new paradigm is proposed, in which the data owner fully outsources both the data storage and the management of the data. In traditional distributed RDC schemes, the repair phase imposes a significant burden on the client, who needs to expend a significant amount of computation and communication, thus, it is very difficult to keep the client lightweight. A new self-repairing concept is developed, in which the servers are responsible to repair the corruption, while the client acts as a lightweight coordinator during repair. To realize this new concept, two novel RDC schemes, RDC-SR and ERDC-SR, are designed for replication-based distributed storage systems, which enable Server-side Repair and minimize the load on the client side.

Version control systems (VCS) provide the ability to track and control changes made to the data over time. The changes are usually stored in a VCS repository which, due to its massive size, is often hosted at an untrusted CSP. RDC can be used to address concerns about the untrusted nature of the VCS server by allowing a data owner to periodically check that the server continues to store the data. The RDC-AVCS scheme being designed relies on RDC to ensure all the data versions are retrievable from the untrusted server over time. The RDC-AVCS prototype built on top of Apache SVN only incurs a modest decrease in performance compared to a regular (non-secure) SVN system.

NEW DIRECTIONS FOR REMOTE DATA INTEGRITY CHECKING OF CLOUD STORAGE

by
Bo Chen

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2014

APPROVAL PAGE

NEW DIRECTIONS FOR REMOTE DATA INTEGRITY CHECKING OF CLOUD STORAGE

Bo Chen

Dr. Reza Curtmola, Dissertation Advisor — Date
Associate Professor of Computer Science, NJIT

Dr. Cristian Borcea, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Guiling Wang, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Nirwan Ansari, Committee Member — Date
Distinguished Professor of Electrical and Computer Engineering, NJIT

Dr. Ian Molloy, Committee Member — Date
Research Staff Member, IBM T. J. Watson Research Center

BIOGRAPHICAL SKETCH

Author: Bo Chen

Degree: Doctor of Philosophy

Date: August 2014

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, New Jersey, 2014

- Master of Engineering in Computer Science,
  Graduate University of Chinese Academy of Sciences, Beijing, China, 2008

- Bachelor of Engineering in Computer Science,
  University of Science and Technology Beijing, Beijing, China, 2005

Major: Computer Science

Presentations and Publications:

Bo Chen and Reza Curtmola, "Remote Data Integrity Checking with Server-Side Repair," journal article in preparation.

Bo Chen and Reza Curtmola, "Auditable Version Control Systems," the 21th Annual Network and Distributed System Security Symposium (NDSS '14), San Diego, CA, USA, February 2014.

Bo Chen and Reza Curtmola, "Towards Self-Repairing Replication-Based Storage Systems Using Untrusted Clouds," The Third ACM Conference on Data and Application Security and Privacy (CODASPY '13), San Antonio, TX, USA, pp. 377-388, February 2013.

Bo Chen and Reza Curtmola, "POSTER: Robust Dynamic Remote Data Checking for Public Clouds," The 19th ACM Conference on Computer and Communications Security (CCS '12), Raleigh, NC, USA, pp. 1043-1045, October 2012.

Bo Chen and Reza Curtmola, "Robust Dynamic Provable Data Possession," The Third International Workshop on Security and Privacy in Cloud Computing (ICDCS-SPCC '12), Macau, China, pp. 515-525, June 2012

Bo Chen and Reza Curtmola, "Robust Dynamic Remote Data Checking for Public Clouds," The 35th IEEE Sarnoff Symposium, Newark, NJ, USA, May 2012

Bo Chen, Reza Curtmola, Giuseppe Ateniese, and Randal Burns, "Remote Data Checking for Network Coding-based Distributed Storage Systems," The Second ACM Cloud Computing Security Workshop (CCSW '10), Chicago, IL, USA, pp. 31-42, October 2010

I dedicate this thesis to:

=====

my beloved ***Jun Dai***

my father ***Naiqiang Chen*** *and my mother* ***Yaqun Che***

my elder brother ***Liang Chen*** *and his wife* ***Yuyan Chang*** *and my little nephew* ***Xiangyu Chen***

=====

ACKNOWLEDGEMENTS

I first want to express my deepest thanks to my advisor, Professor Reza Curtmola. Reza initially inspired me to do research in the field of applied cryptography and cloud storage security. Throughout the years, he worked closely with me, providing me a great deal of invaluable guidance and advice on both my research and my life. He helped me to improve my reading, writing, presenting, and critical thinking, which prepared me for becoming a qualified researcher. He also offered me a lot of good opportunities to go to academic events such that I was able to learn from other researchers as well as disseminate our results to the research community.

I also want to express my deep thanks to Professor Cristian Borcea, Professor Guiling Wang, Professor Nirwan Ansari, and Dr. Ian Molloy. They served on my PhD dissertation committee, and were dedicated to helping me to improve my dissertation. I benefited a lot from their invaluable advice and suggestions.

Special thanks to Professor Giuseppe Ateniese and Professor Randal Burns. I was so lucky to collaborate with them in the "network coding-based storage systems" project.

During my graduate studies, I was very lucky to receive invaluable advice regarding to my research and (or) my future career from the following people: Professor Reza Curtmola, Professor Radu Sion, Professor Cristina Nita-Rotaru, Professor Nirwan Ansari, Professor Guiling Wang, and Professor Meng Yu.

I would like to thank Professor Reza Curtmola, Professor Marvin Nakayama, Professor James Calvin, Professor Boris Verkhovsky, and Professor Andrew Sohn for advising me in teaching when I was a teaching assistant.

I would like to thank the professors who supervised my graduate courses: Reza Curtmola, Cristian Borcea, Guiling Wang, David Nassimi, Alexandros Gerbessiotis, Dimitrios Theodoratos, Joseph Leung, Yehoshua Perl, James Geller, and Chengjun Liu.

During the past few years, I was very lucky to have the opportunities to visit other research institutes and present my work. Those would not be possible without the help of the following people: Dr. Ian Molloy, Professor Yujun Zhang, Professor Hanwen Zhang, Professor Zhan Wang, Dr. Xinchang Zhang, Professor Georgios Portokalidis.

I worked closely with the following students in my research, including: Ying Chen, Arthur Hinds, Kirtan Shah, Anilkumar Ammula. I would like to thank them for their invaluable contributions to the collaborative work.

I owe everything to my beloved Jun Dai, my parents and my elder brother. Without their love and continuous support, it would have been tough for me to get through all the difficulties in my everlasting studies. I also want to thank my cousin Hong Lin who offered a lot of help to my parents in the past few years when I was far away in the United States.

The life in US was wonderful and interesting thanks to the help of my dear friends and student colleagues. I would like to express my gratitude to them, including: Yang Li, Zhiying Qiu, Tan Yan, Yiyi Wu, Jie Tian, Yaqiong Zhao, Suan Pan, David Paglia, Xiaoyuan Liang, Xin Gao, Xin Xu, Xian Wu, Manoop Talasila, Hillol Debnath, Nafize Paiker, Mohammad Ashrafuzzaman Khan, Daniel Boston, Pei Li, Xiangyi Kong, Jinglin Jiang, Xi Chen, JLurker Kao, IMing Lee, Xian Hu, Wei Zhang, Shuo Chen, Zhiming Liu, Duo Wei, Xinfa Hu, Xiaoying Wu, Yuan Yuan, Zhe He, Fang Chu, Wei Xiong, Lynn Greiner, Xiangqian Yu, Ye Tian, Jinhui Zheng, Siyuan Lv, Na Li, Haitao Xu, Mengran Xu, Shouxian Cheng, Man Zhang, Yong Zhang, Huigen Zhang, Jianhua Zhou.

Also thanks to the administrative staffs from the Computer Science Department, Office of Graduate Studies and Office of International Students including: Dr. George Olsen, Ms. Kathy James, Ms. Angel Bell, Dr. Sotirios Ziavras, Ms. Clarisa Gonzalez-Lenahan, Mr. Jeffrey Grundy.

The financial support for my graduate studies was possible thanks to the teaching assistantship from the Department of Computer Science, NJIT and the NSF grants CAREER 1054754-CNS and 1241976-DUE.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Outsourcing is a popular business model nowadays. A traditional way of outsourcing involves transferring employees and assets from one firm to another [1]. In recent years, a novel way of outsourcing – IT infrastructures outsourcing – has been developed. This brand-new outsourcing model involves transferring IT infrastructures from one firm who cannot budget too much on IT expenses, to another who has competencies and expertise on IT management and maintenance. Cloud computing is the core technology supporting this new outsourcing model. Cloud service providers like Amazon Web Services [2] and Windows Azure [3] allow users (*e.g.*, firms and organizations) to outsource their IT infrastructures, and simply pay for what they have used (pay as you go). In this way, cloud users can save their investments on the fixed assets of IT infrastructures, and thus may greatly reduce their overall IT expenses. Besides low cost, cloud computing offers great flexibility, good scalability, and high reliability, which make it a top technology priority for chief information officers [4].

Cloud storage is one of the most prevalent cloud services. With the deployment of cloud storage services (*e.g.*, Amazon S3 [5] and Glacier [6], Azure cloud storage [3], etc.), data owners can choose to outsource their data to the cloud storage providers (CSP), such that they can get liberated from the burden of both data storage and management. Although cloud storage has many benefits, most of the data owners today are reluctant to outsource their data, simply because they do not trust the CSPs: they are not sure whether their valuable data will be correctly maintained and protected by the CSPs; they are not

sure whether they can correctly retrieve their data over time. A gap thus arises between a cloud's benefits and its security breaches. To bridge this gap, a question is posed: **Is it possible to allow data owners to outsource their data to the cloud and, at the same time, obtain similar security guarantees like when the data is stored in their own data centers?** In this dissertation, we try to answer this question by investigating remote data checking (RDC), a technique which can allow data owners to obtain guarantees that their outsourced data is retrievable over time. In general, a data owner may choose to delegate the workload of obtaining such a guarantee to a third party, which is termed as "a verifier" and "an auditor" interchangeably throughout the dissertation. In the following, we provide an overview for the entire dissertation.

1.1 Remote Data Checking

Remote data checking allows a client (data owner) to check the integrity of data outsourced at an untrusted server, and thus to audit whether the server fulfills its contractual obligations. The ultimate goal of remote data checking is to ensure that data owners will be able to recover the same exact data they have stored in the cloud. A basic RDC protocol consists of three phases: Setup, Challenge and Retrieve. During Setup, the data owner preprocesses the file $\mathtt{F}$ generating metadata Σ, and then stores both $\mathtt{F}$ and Σ at the server. The data owner deletes $\mathtt{F}$ and Σ from its local storage and only keeps a small amount of secret key material K (**constant client storage**). During Challenge, an auditor (the data owner or another client) challenges the server to prove that it can produce the data that was originally stored by the data owner. The server produces a proof of data possession based on the stored data and metadata. The client can then use the secret key material K to check

the validity of the proof provided by the server. During Retrieve, the data owner recovers the original data.

In general, both the Setup and the Retrieve are rare events, but the Challenge happens periodically. Thus, how to minimize the cost of the Challenge phase is a main concern of an efficient RDC scheme. In the literature, a technique based on spot checking – the auditor only checks a random subset of the whole data – is adopted to minimize the cost of the Challenge phase. Previous result [7] shows that, if the attacker corrupts a certain amount of the whole data (*e.g.*, 1%), the auditor can detect such corruptions with high probability by only randomly checking a constant number of data blocks.

1.2 Dynamic Remote Data Checking

Early RDC schemes have focused on *static* data, in which the client cannot modify the original data [7–10] or can only perform a limited set of updates [11]. Later work [12–15] extends RDC to support the full range of dynamic operations with optimal cost.

Dynamic Provable Data Possession (DPDP) [12] proposes a model that provides strong guarantees about data integrity while supporting the full range of dynamic operations on the outsourced data, including modifications, insertions, deletions, and appends. A DPDP protocol contains the three phases as in an RDC protocol for static data (Setup, Challenge, and Retrieve), but also allows another phase, Update. During Update, the original file may be changed by insertions, deletions, modifications, and appends. During Challenge, the auditor obtains an integrity guarantee about the latest version of the file (due to updates, this may be different from the original file). In Retrieve, the client recovers the latest file version. As opposed to handling static data, the main challenge in DPDP is ensuring that the client obtains a guarantee about the latest version of the file (*i.e.*, prevent

the server from passing the client's challenges by using old file versions) while meeting the low overhead requirements for RDC.

1.3 Robust Remote Data Checking

Remote data checking schemes (static or dynamic) usually adopt spot checking technique for efficiency. However, it is very difficult for spot checking to detect corruption of small parts of the data, *e.g.*, 1 byte. Robustness is thus proposed to supplement RDC for the small corruption concern. A *robust* remote data checking scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption, in which a notion of mitigation should include the ability to both efficiently detect data corruption and be impervious to data corruption. When data corruption is detected, the owner can act in a timely fashion (*e.g.*, data can be repaired from other replicas). Even when data corruption (*e.g.*, small corruption) is not detected, a robust auditing scheme ensures that no data will be lost, *i.e.*, robustness guarantees that small corruptions which cannot be detected (*e.g.*, by spot checking) can always be recovered.

In general, error correcting codes can be used to add robustness to a remote data checking scheme. The challenge is how to efficiently integrate error correcting codes with RDC to achieve robustness guarantees. Prior work [8, 16, 17] investigates how to add robustness to a static RDC scheme, but it is still an open problem to add robustness to a dynamic RDC scheme with small, constant, client storage. In Chapter 2, we design R-DPDP (Robust Dynamic Provable Data Possession), an RDC scheme that provides robustness and, at the same time, supports dynamic updates, while requiring small, constant, client storage. We propose two R-DPDP constructions. The first construction πR-D achieves robustness by extending techniques from the static to the

dynamic setting. The resulting R-DPDP construction is efficient in encoding, but requires a high communication cost for updates (insertions/deletions). The second construction, VLCG (Variable Length Constraint Group), overcomes this drawback by: (a) decoupling the encoding for robustness from the position of symbols in the file and instead relying on the value of symbols, and (b) reducing expensive insert/delete operations to append/modify operations when updating the RS-coded parity data, which ensures efficient updates even under an adversarial setting.

1.4 Remote Data Checking for Distributed Settings

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

When a distributed storage system is used in tandem with remote data checking, one can distinguish several phases throughout the lifetime of the storage system: Setup, Challenge, and Repair. To outsource a file, the data owner encodes the file by introducing redundancy during Setup and distributes the encoded data to multiple storage servers.

During the Challenge phase, the data owner can ask periodically each server to provide a proof that the server's stored data has remained intact. If a server is found corrupted during the Challenge phase, the data owner can take actions to Repair it relying on the data from the healthy servers, thus restoring the desired redundancy level in the system.

The main approaches to introduce redundancy in distributed storage systems are through *replication*, *erasure coding*, and more recently through *network coding* [18, 19]. The basic principle of data replication is to store multiple copies of the data at different storage servers, whereas in erasure coding the original data is encoded into fragments which are stored across multiple storage servers. Previous work [20] shows that erasure codes can achieve equivalent or even better reliability level than replication by significantly lower storage overhead. Network coding for storage [18, 19] provides nice performance properties well suited to deep archival stores which are characterized by a *read-rarely* workload. Similar to erasure coding, network coding can be used to redundantly encode a file into fragments and store these fragments at multiple servers. Network coding provides a *significant advantage* over erasure coding when coded fragments are lost due to server failures and need to be reconstructed in order to maintain the same level of reliability. Previously, network coding-based distributed storage systems have been investigated in the benign setting [18, 19]. We are the first to consider remote data checking for network coding-based distributed storage systems which rely on untrusted servers (*i.e.*, an adversarial setting). In Chapter 3, we identify new attacks and propose RDC-NC, a novel Remote Data Checking scheme for Network Coding-based distributed storage systems. We adapt RDC techniques used in the single server setting [10] to handle data corruption attacks and collusion attacks (among malicious servers). We also identify the *replay* and *pollution* attacks and come up with effective solutions. To handle replay attacks, we encrypt

the coding coefficients which are stored on the servers; moreover, the client is the one that chooses the coding coefficients and enforces their use. To prevent pollution attacks, we use an additional repair verification tag, which allows the client to check that a server combines its blocks correctly during the repair phase.

The setting considered so far outsources the storage of the data, but the data owner is still heavily involved in the data management process (especially during the repair of damaged data). A new paradigm is thus investigated, in which the data owner fully outsources both the data storage and the management of the data, *i.e.*, 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 traditional distributed RDC schemes [21–23], the repair phase imposes a significant burden on the client, who needs to expend a significant amount of computation and communication. For example, to repair the 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. Archival storage deals with large amounts of data (Terabytes or even Petabytes) and thus maintaining the health of the data imposes a heavy burden on the data owner. We work on a new concept, namely, server-side repair, in which the servers are responsible to repair the corruption, while the client acts as a lightweight coordinator during repair. We propose two novel RDC schemes for replication-based distributed storage systems, RDC-SR (Chapter 4) and ERDC-SR (Chapter 5), which enable server-side repair and minimize the load on the client (data owner) side. In both schemes, servers are allowed to collaborate in order to generate a new replica whenever a replica has failed. However, this comes at the cost of allowing a new attack avenue for servers, the ROTF attack. To overcome the ROTF (Replicate On The Fly) 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. Although they try to achieve a similar objective, RDC-SR and ERDC-SR are different in that, RDC-SR assumes that the computational power of the CSP will not grow over time, whereas ERDC-SR relaxes this assumption and is thus more suitable for real-world applications.

1.5 Remote Data Checking for Version Control Systems

Version control provides the ability to track and control changes made to the data over time. Software development often relies on a Version Control System (VCS) to automate the management of source code, documentation and configuration files. The VCS system stores all the changes to the data into a repository, such that any version of the data can be retrieved at any time in the future. Due to their potentially massive size, VCS repositories are often hosted at the untrusted CSPs. Remote data checking can thus be used to address concerns about the untrusted nature the VCS server by allowing a data owner to periodically and efficiently check that the server continues to store the data.

To reduce the storage overhead, modern version control systems usually adopt "delta encoding", in which only the differences (between versions) are recorded. As a particular type of delta encoding, skip delta encoding can optimize the combined cost of storage and retrieval.

In Chapter 6, we introduce *Auditable Version Control Systems* (AVCS), which are VCS systems designed to function under an adversarial setting. We present the definition of AVCS and then propose RDC-AVCS, an AVCS scheme for skip delta-based VCS systems, which relies on RDC mechanisms to ensure all the versions of a file are retrievable from the untrusted VCS server over time. In RDC-AVCS, the cost of checking the integrity

of all the versions of a file is the same as checking the integrity of one file version and the client is only required to maintain the same amount of client storage like a regular (non-secure) VCS system. We make the important observation that the only meaningful operation for real-world VCS systems which use delta encoding is *append* and leverage this observation to build RDC-AVCS. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, we take a pragmatic approach and provide a solution for real-world VCS systems.

1.6 Organization

Portions of this dissertation are drawn from the following publications:

- "Robust dynamic provable data possession" [24]
- "POSTER: Robust dynamic remote data checking for public clouds" [25]
- "Remote data checking for network coding-based distributed storage systems" [23]
- "Towards self-repairing replication-based storage systems using untrusted clouds" [26]
- "Auditable Version Control Systems" [27]

In order to allow data owners to obtain security guarantees of the correctness and retrievability of the outsourced data, RDC mechanisms have been investigated extensively in the literature. In Figure 1.1, we provide a diagram for the RDC literature, which shows how the RDC protocols designed in this dissertation fit into the RDC literature. These newly designed RDC protocols are organized throughout the entire dissertation as follows:

In Chapter 2, we work on the robustness issue of dynamic RDC. We design a robust dynamic provable data possession scheme (R-DPDP, Section 2.3), which mainly relies on the interesting properties of Cauchy Reed-Solomon codes (Section 2.2). We provide

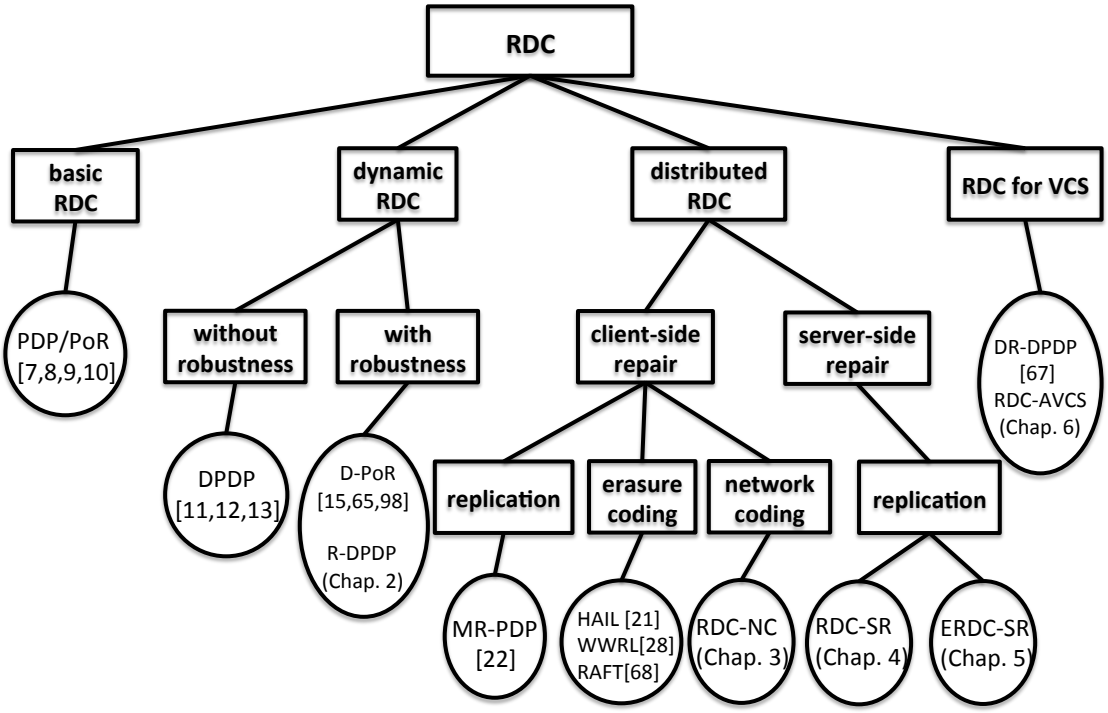


Figure 1.1 A diagram summarizing the Remote Data Checking (RDC) literature.

security and performance analysis for R-DPDP (Section 2.4), and discuss solutions for further optimization (Section 2.5).

In Chapter 3, we devise RDC-NC, an RDC scheme for network coding-based distributed storage systems (Section 3.4). We provide security analysis in Section 3.5. We also provide guidelines on selecting the parameters (Section 3.6) and experimental evaluations (Section 3.7).

We design two RDC schemes, RDC-SR (Chapter 4) and ERDC-SR (Chapter 5), for replication-based storage systems which can support server-side repair. For RDC-SR, we enhance a network delay-based model (Section 4.2), and adapt it to the design of RDC-SR scheme (Section 4.4). We provide guidelines (Section 4.5) on how to use RDC-SR, analyze

its security (Section 4.6), and build a prototype on top of Amazon Cloud (Section 4.7). For ERDC-SR, we elaborate its design (Section 5.2), as well as provide guidelines for using it in practical applications (Section 5.3). We also provide security (Section 5.4) and performance analysis (Section 5.5).

In Chapter 6, we introduce RDC-AVCS, an AVCS (Section 6.5.2) scheme for skip delta-based version control systems. We provide the background of delta-based version control systems in Section 6.3, and elaborate the designed scheme in Section 6.5. Security analysis of RDC-AVCS (Section 6.6) and its prototype implementation on top of Apache SVN (Section 6.7) are also provided.

We conclude in Chapter 7.

1.7 Notations

Acronyms used throughout the entire dissertation are shown in Table 1.1.

Table 1.1 Acronyms

RDC	Remote Data Checking
PDP	Provable Data Possession
PoR	Proofs of Retrievability
DPDP	Dynamic Provable Data Possession
D-PoR	Dynamic Proofs of Retrievability
VLCG	Variable Length Constraint Group
CSP	Cloud Storage Provider
R-DPDP	Robust Dynamic Provable Data Possession
RS	Reed-Solomon
RDC-NC	Remote Data Checking for Network Coding-based distributed storage systems
ROTF	Replicate On The Fly
RDC-SR	a replication-based RDC scheme with Server-side Repair
ERDC-SR	an Enhanced replication-based RDC scheme with Server-side Repair
VCS	a Version Control System
AVCS	an Auditable Version Control System
RDC-AVCS	an Auditable Version Control System based on Remote Data Checking
SVN	Subversion
SSVN	Secure SVN
APR	Apache Portable Runtime

CHAPTER 2

ROBUST DYNAMIC PROVABLE DATA POSSESSION

This chapter introduces R-DPDP, a robust dynamic provable data possession scheme. R-DPDP is the first RDC scheme that provides robustness and, at the same time, supports dynamic data updates, while requiring small, constant, client storage. R-DPDP initiates the research of D-PoR (Dynamic Proofs of Retrievability), which received significant coverage in the literature.

Remote Data Checking (RDC) is a technique that allows to check the integrity of data stored at a third party, such as a Cloud Storage Provider (CSP). Especially when the CSP is not fully trusted, RDC can be used for data auditing, allowing data owners to assess the risk of outsourcing data in the cloud.

In an RDC protocol, the data owner (client) initially stores data and metadata with the cloud storage provider (server); at a later time, an auditor (the data owner or another client) can challenge the server to prove that it can produce the data that was originally stored by the client; the server then generates a proof of data possession based on the data and the metadata. Several RDC schemes have been proposed, including Provable Data Possession (PDP) [7, 8] and Proofs of Retrievability (PoR) [9, 10], both for the single server [7, 9, 10] and for the multiple server setting [21–23, 28].

Early RDC schemes have focused on *static* data, in which the client cannot modify the original data [7, 9, 10] or can only perform a limited set of updates [11]. Erway et al. [12] have proposed DPDP, a scheme that supports the full range of *dynamic updates* on the outsourced data, while providing the same strong guarantees about data integrity.

The ability to perform updates such as insertions, modifications, or deletions, extends the applicability of RDC to practical systems for file storage [29, 30], database services [31], peer-to-peer storage [32, 33], and more complex cloud storage systems [34, 35].

A scheme for auditing remote data should be both *lightweight* and *robust* [8]. *Lightweight* means that it does not unduly burden the server; this includes both overhead (*i.e.*, computation and I/O) at the server and communication between the server and the client. This goal can be achieved by relying on *spot checking*, in which the client randomly samples small portions of the data and checks their integrity, thus minimizing the I/O at the server. Spot checking allows the client to detect if a fraction of the data stored at the server has been corrupted, but it cannot detect corruption of small parts of the data (*e.g.*, 1 byte).

Robust means that the auditing scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption. Protecting against **large corruptions** ensures the CSP has committed the contracted storage resources: Little space can be reclaimed undetectably, making it unattractive to delete data to save on storage costs or sell the same storage multiple times. Protecting against **small corruptions** protects the data itself, not just the storage resource. Many data have value well beyond their storage costs, making attacks that corrupt small amounts of data practical. For example, modifying a single bit may destroy an encrypted file or invalidate authentication information. Thus, *robustness is a necessary property* for all RDC schemes that rely on spot checking, which includes the majority of static and dynamic RDC schemes.

Robustness is usually achieved by integrating forward error-correcting codes (FECs) with remote data checking [8, 16, 17]. Attacks that corrupt small amounts of data do no damage, because the corrupted data may be recovered by the FEC. Attacks that do unrecoverable amounts of damage are easily detected using spot checking, because they

must corrupt many blocks of data to overcome the FEC redundancy. Unfortunately, under an adversarial setting, there is a fundamental tension between the dynamic nature of the updates supported in the DPDP scheme and FEC codes (which are mostly designed for static data) because securely updating even a small portion of the file may require retrieving the entire file.

On the Adversarial Model. In this work, the cloud storage server is assumed to be not trustworthy. Protection against corruption of a large portion of the data is necessary in order to handle servers that discard a significant fraction of the data. This applies to servers that are financially motivated to sell the same storage resource to multiple clients.

Protection against corruption of a small portion of the data is necessary in order to handle servers that try to hide data loss incidents. This applies to servers that wish to preserve their reputation. Data loss incidents may be accidental (*e.g.*, management errors or hardware failures) or malicious (*e.g.*, insider or outsider attacks).

Moreover, the storage server may try to provide a stale, older version of the data.

2.1 Background and Related Work

2.1.1 Remote Data Checking for Dynamic Settings

Early RDC schemes have focused on *static* data, in which the client cannot modify the original data [7, 9, 10] or can only perform a limited set of updates [11]. **Dynamic Provable Data Possession (DPDP)** [12] proposes a new RDC model that provides strong guarantees about data integrity while supporting the full range of dynamic operations on the outsourced data, including modifications, insertions, deletions, and appends. A DPDP protocol contains four phases: Setup, Update, Challenge, and Retrieve. During Setup, the client preprocesses the file generating the metadata, and then stores in the server both the

file and the metadata. The client can now delete the data and metadata locally to save storage. During Update, the original file may be updated. During Challenge, the auditor obtains an integrity guarantee about the latest version of the file, which may be different from the original file due to updates. During Retrieve, the client retrieves the latest version of the file. As opposed to handling static data, the main challenge in DPDP is ensuring that the client obtains guarantees about the latest version of the file (*i.e.*, prevent the server from passing the client's challenges by using old file versions) while meeting the low overhead requirements for RDC.

A DPDP scheme is a collection of seven polynomial-time algorithms (KeyGen_DPDP, PrepareUpdate_DPDP, PerformUpdate_DPDP, VerifyUpdate_DPDP, GenChallenge_DPDP, Prove_DPDP, Verify_DPDP) that can be used to construct a DPDP protocol as follows. During the Setup phase, the client uses KeyGen_DPDP to setup the scheme and PrepareUpdate_DPDP to preprocess the file and generate metadata. The server stores the client's data using PerformUpdate_DPDP and the client uses VerifyUpdate_DPDP to check the success of the initial file submission (note that the initial file submission can be seen as an update in which the client re-writes the entire file). In the Update phase, the client and server use PrepareUpdate_DPDP, PerformUpdate_DPDP and VerifyUpdate_DPDP to prepare the update, apply the update on the file, and verify if the update was applied correctly, respectively. During the Challenge phase, the client uses GenChallenge_DPDP to generate a challenge, the server generates a proof of data possession using Prove_DPDP, and the client verifies the proof using Verify_DPDP.

A complete definition of a DPDP scheme is provided in Appendix A.1, which does not include provisions for robustness.

Dynamic Proofs of Retrievability. Concurrently with our work, Stefanov et al. [15] proposed Iris, a system that supports dynamic proofs of retrievability (D-PoR), including protection against small data corruption. For practical reasons, Iris achieves robustness by storing the parity data on the client. As this may place an additional burden on lightweight clients, our work focuses on a more challenging setting which has stood as an open problem: All data, including parity, is stored at the server, in order to minimize client storage. Cash et al. [36] propose to use Oblivious RAM to construct a D-PoR scheme. However, Oblivious RAM is too expensive to be used in a practical application. Another proposal for D-PoR [14] does not offer protection against small data corruption when clients rely on spot checking data stored at untrusted servers.

Authenticated Data Structures. In all the DPDP and D-PoR constructions, the client uses an *authenticated data structure* to ensure the freshness of the retrieved file and to prevent the server from using an old file version when answering challenges. This data structure is usually a tree-like structure computed over the verification tags, and the client keeps a copy of the root of this structure (*e.g.*, skip lists [12], RSA trees [12], Merkle hash trees [13, 15], or 2-3 trees [14]). Our work can rely on any of these data structures to ensure file data freshness and prevent the server from conducting replay attacks.

2.1.2 Robust Auditing of Outsourced Data

A *robust* auditing scheme incorporates mechanisms for mitigating arbitrary amounts of data corruption. A notion of mitigation should include the ability to both efficiently detect data corruption and be impervious to data corruption. When data corruption is detected, the owner can act in a timely fashion (*e.g.*, data can be restored from other replicas). Even when

data corruption (*e.g.*, small corruption) is not detected, a robust auditing scheme ensures that no data will be lost. More formally, a robust auditing scheme is defined as follows [8]:

Definition 2.1.1. *A* robust auditing scheme $\mathcal{RA}$ *is a tuple* $(\mathcal{C}, \mathcal{T})$*, where* $\mathcal{C}$ *is a remote data checking scheme for a file* $\mathtt{F}$ *and* $\mathcal{T}$ *is a transformation that yields* $\tilde{\mathtt{F}}$ *when applied on* $\mathtt{F}$*.* $\mathcal{RA}$ *is considered to provide* δ-robustness *when:*

– the auditor will detect *with high probability if the server corrupts more than a* δ*-fraction of* $\tilde{\mathtt{F}}$ *(**protection against corruption of a** **large** **portion of** $\tilde{\mathtt{F}}$**)***
– the auditor will recover *the data in* $\mathtt{F}$ *with high probability if the server corrupts at most a* δ*-fraction of* $\tilde{\mathtt{F}}$ *(**protection against corruption of a** **small** **portion of** $\tilde{\mathtt{F}}$**)***

δ-robustness guarantees that small corruptions (less than a δ fraction of the whole data) which cannot be detected (*i.e.*, by spot checking) can always be recovered. Several methods can be employed to add robustness to a static remote data checking scheme [8,16]. The most straightforward method is to use an FEC code over the entire file. For a file of f symbols, this can be achieved with an (n, f) Reed-Solomon code and would give an even stronger guarantee than δ-robustness (Section 1.3), because this code can deterministically correct up to $n - f$ erasures and not just with high probability. Such an FEC code would be impractical because RS codes become quite inefficient to compute even for moderate-size files if the code were to be applied over the entire file.

For efficiency reasons, it is desirable to apply an RS code over a smaller number of symbols: The file $\mathtt{F}$ is divided into k-symbol chunks and a (n, k) RS code is applied to each chunk, expanding it into an n-symbol code word. The first k symbols of the code word are the original k data symbols, followed by $d = n - k$ parity symbols. A *constraint group* is defined as the group of symbols from the same code word, *i.e.*, the original k data symbols and their corresponding $n - k$ parity symbols. The number of constraint groups in the encoded file is the same as the number of chunks in the original file, namely, $\frac{f}{k}$.

To ensure the δ-robustness guarantee, **it is necessary that the association between symbols and constraint groups remain hidden** (*i.e.*, the malicious server should not know which symbols belong to the same constraint group, so that it cannot make the data from this constraint group unrecoverable by only deliberately corrupting a small portion of the data from this constraint group, which is always smaller than a δ fraction of the whole file). This can be achieved through a combination of permuting and then encrypting the symbols of the file. Different encoding schemes to add robustness can lead to remote data checking schemes with different properties and performance characteristics [8, 9, 16, 17]. Two of them are reviewed in the following.

Let $(\mathsf{G}, \mathsf{E}, \mathsf{D})$ be a symmetric-key encryption scheme and π, ψ, ω be pseudo-random permutations (PRPs) defined as:

- $\pi : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(fn/k)} \rightarrow \{0,1\}^{\log_2(fn/k)}$
- $\psi : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(f)} \rightarrow \{0,1\}^{\log_2(f)}$
- $\omega : \{0,1\}^{\kappa} \times \{0,1\}^{\log_2(fd/k)} \rightarrow \{0,1\}^{\log_2(fd/k)}$

The keys w, z, v, u are used for the encryption scheme, PRP π, PRP ψ and PRP ω, respectively.

Permute-All (πA). The constraints among symbols can be concealed by randomly permuting and then encrypting *all* the symbols of the encoded file. Starting from the file $\mathtt{F} = \mathtt{b}_1, \ldots, \mathtt{b}_f$, an (n, k) RS code (with $d = n - k$) is used to generate the encoded file $\hat{\mathtt{F}} = \mathtt{b}_1, \ldots, \mathtt{b}_f, \mathtt{c}_1, \ldots, \mathtt{c}_{\frac{f}{k}d}$, in which symbols $\mathtt{b}_{ik+1}, \ldots, \mathtt{b}_{(i+1)k}$ are constrained by parity symbols $\mathtt{c}_{id+1}, \ldots, \mathtt{c}_{(i+1)d}$, for $0 \leq i \leq \frac{f}{k} - 1$. π and E are used to randomly permute and then encrypt all the symbols of $\hat{\mathtt{F}}$, obtaining the encoded file $\tilde{\mathtt{F}}$, where $\tilde{\mathtt{F}}[i] = \mathsf{E}_w(\hat{\mathtt{F}}[\pi_z(i)])$, for $1 \leq i \leq fn/k$.

This strategy leads to a δ-robustness guarantee [8, 9, 16], but has two major drawbacks: Permuting the entire encoded file can be inefficient and the systematic nature of the RS code is sacrificed.

Permute-Redundancy (πR). The drawbacks of the πA scheme can be overcome based on the observation that it is sufficient to *permute and then encrypt only the parity symbols*. The input file $\mathtt{F} = \mathtt{b}_1, \ldots, \mathtt{b}_f$ is encoded as follows:

1. Use ψ to randomly permute the symbols of $\mathtt{F}$ to obtain the file $\mathtt{P} = \mathtt{p}_1, \ldots, \mathtt{p}_f$, where $\mathtt{p}_i = \mathtt{b}_{\psi_v(i)}$, $1 \leq i \leq f$.
2. Compute parity symbols $\mathtt{C} = \mathtt{c}_1, \ldots, \mathtt{c}_{\frac{f}{k}d}$ so that symbos $\mathtt{p}_{ik+1}, \ldots, \mathtt{p}_{(i+1)k}$ are constrained by $\mathtt{c}_{id+1}, \ldots, \mathtt{c}_{(i+1)d}$, for $0 \leq i \leq \frac{f}{k} - 1$.
3. Permute and then encrypt the parity symbols to obtain $\mathtt{R} = \mathtt{r}_1, \ldots, \mathtt{r}_{\frac{f}{k}d}$, where $\mathtt{r}_i = \mathsf{E}_w(\mathtt{c}_{\omega_u(i)})$, $1 \leq i \leq \frac{f}{k}d$.
4. Output redundancy encoded file $\tilde{\mathtt{F}} = \mathtt{F}||\mathtt{R}$.

By computing RS codes over the permuted input file, rather than the original input file, an attacker does not know the relationship among symbols of the input file. By permuting the parity symbols, the attacker does not know the relationship among the symbols in the redundant portion $\mathtt{R}$ of the output file. By encrypting the parity symbols, an attacker cannot find the combinations of input symbols that correspond to output symbols.

When compared to πA, πR is more efficient (as it requires to permute and encrypt only the parity symbols) and preserves the systematic nature of the RS code. πR was shown to achieve the δ-robustness guarantee [16, 17] (this construction is also known as a "server code" in the literature [17, 21]).

2.2 Cauchy Reed-Solomon Codes

Towards achieving robustness for dynamic RDC schemes, in this section we first review Reed-Solomon encoding and decoding based on Cauchy matrices. We then study how to update a Reed-Solomon code when an update is applied to the original data. Note that in this section we study these operations under a benign setting (*i.e.*, when the server is trustworthy).

We consider an (n, k) Reed-Solomon (RS) code that can correct up to $d = n - k$ known erasures or $\lfloor \frac{d}{2} \rfloor$ unknown errors, or any combination of E errors and S erasures with $2E + S \leq d$. The minimum Hamming distance of the RS code is $d + 1$, where $d = n - k$. If two RS codes have the same value d, we say they provide *the same fault tolerance level*.

To encode a k-symbol message into an n-symbol code word, we need an $n * k$ encoding matrix, known as the *distribution matrix*. Typically, Vandermonde or Cauchy matrices are used to construct the distribution matrix. We use Cauchy RS codes, which are Reed-Solomon codes based on Cauchy matrices [37], for two reasons: They are more suitable to handle dynamic operations on the original data and they were shown to be approximately twice as fast as the classical Reed-Solomon encoding based on Vandermonde matrices [38–41].

2.2.1 Cauchy RS Encoding and Decoding

For ease of presentation, we present the Cauchy RS encoding and decoding using a (6,4) RS code as an example (*i.e.*, $n = 6, k = 4, d = 2$). All the arithmetic operations are in $\mathbb{F}_{2^w}$, assuming the condition $2^w > n$ always holds (the $+$, $-$ operations can be regarded as $\oplus$, logical XOR). We use $\mathbf{L}^{\mathrm{T}}$ to denote the transpose of a vector $\mathbf{L}$.

Encode. The message $\mathbf{L}$ contains 4 data symbols, all of which are in the Galois Field $\mathbb{F}_{2^w}$: $\mathbf{L} = (\ b_1 \quad b_2 \quad b_3 \quad b_4\ )$.

We use the method introduced in [42] to construct the Cauchy matrix, which has the useful property that it can be re-generated on the fly based on a constant amount of information. The distribution matrix M_1, which is composed of the identity matrix in the first 4 rows and Cauchy matrix in the remaining 2 rows, is as follows:

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, \text{where } a_{ij} = \frac{1}{i \oplus (d+j)}$$

The codeword C is computed as

$$C = M_1 \times \mathbf{L}^{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ p_1 \\ p_2 \end{pmatrix}$$

where the parity symbols p_1 and p_2 are

$$p_1 = a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4$$

$$p_2 = a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4$$

Decode. When using a $(6, 4)$ RS code, any 4 out of 6 symbols are enough to recover $\mathbf{L}$. Assume that b_3 and b_4 are corrupted. To recover $\mathbf{L}$, the decoding process is:

$$\mathbf{L}^{\mathrm{T}} = M^{-1} \times \begin{pmatrix} b_1 \\ b_2 \\ p_1 \\ p_2 \end{pmatrix}, \text{where } M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

The decoding matrix, M, is invertible based on the fact that all of the $4*4$ submatrices of M_1 are invertible [43]. Moreover, M has the useful property that it can be re-generated by knowing the indices of the non-corrupted symbols in the code word. By putting the non-corrupted symbols into their right locations in the code word for decoding, M can be re-generated based on a constant amount of information.

2.2.2 Cauchy RS Updating

Consider a (n, k) Cauchy RS code computed over a message. If the symbols in the original message are updated (*e.g.*, modified, appended, inserted, deleted), we are interested to update the RS parity data so that it reflects the updated message. We seek to answer the question: How can we minimize the cost of updating the RS code? More precisely, how can we update the parity symbols efficiently by minimizing the number of symbols that need to be read from the original RS code?[1]. We answer this question by using the same example introduced from Section 2.2.1. The conclusion is that *modify/append operations have a lower bandwidth overhead than insert/delete operations*.

Modify a data symbol. For example, if symbol b_1 is modified to b'_1, we should update the parity data correspondingly: p_1 is updated to p'_1, and p_2 is updated to p'_2. To compute p'_1 and p'_2, only *the old parity symbols* (p_1, p_2) and *the old data symbol* (b_1) are required to be retrieved (*i.e.*, there is no need to retrieve any other data symbol except the one to be modified):

[1]For the purpose of RDC, the client needs to update the RS code on the server after each update operation, so we are interested in minimizing the data communication required to update the RS code.

$$
\begin{aligned}
p'_1 &= a_{11} * b'_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 \\
&= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 + a_{11} * b'_1 - a_{11} * b_1 \\
&= p_1 + a_{11} * b'_1 - a_{11} * b_1
\end{aligned}
$$

$$
\begin{aligned}
p'_2 &= a_{21} * b'_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 \\
&= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 + a_{21} * b'_1 - a_{21} * b_1 \\
&= p_2 + a_{21} * b'_1 - a_{21} * b_1
\end{aligned}
$$

Append a data symbol. For example, if b_5 is appended to $\mathbf{L}$, to maintain *the same fault tolerance level*, the $(6, 4)$ RS code should become a $(7, 5)$ RS code and the new distribution matrix M_2 is (assuming the condition $2^w > n$ still holds):

$$
M_2 = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25}
\end{pmatrix}, \text{ where } a_{ij} = \frac{1}{i \oplus (d + j)}
$$

Compared to M_1, most of the elements in M_2 are the same although n has changed (this is a useful property of Cauchy RS codes). To update the parity data correspondingly, only *the old parity symbols* (p_1, p_2) are required to be retrieved (*i.e.*, there is no need to retrieve any of the data symbols):

$$
\begin{aligned}
p'_1 &= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 + a_{14} * b_4 + a_{15} * b_5 \\
&= p_1 + a_{15} * b_5
\end{aligned}
$$

$$
\begin{aligned}
p'_2 &= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 + a_{24} * b_4 + a_{25} * b_5 \\
&= p_2 + a_{25} * b_5
\end{aligned}
$$

Insert a data symbol. For example, if b'_2 is inserted into $\mathbf{L}$ after b_2, to maintain *the same fault tolerance level*, the $(6, 4)$ RS code should become a $(7, 5)$ RS code. The new

distribution matrix will be M_2, thus, we can update the parity data (p_1 to p'_1, p_2 to p'_2) by retrieving whichever one is smaller between: (i) *all the data symbols* (*i.e.*, b_1, b_2, b_3, b_4), or (ii) *all the parity symbols* (p_1, p_2) and *the data symbols after* b_2 (*i.e.*, b_3, b_4):

$$\begin{aligned} p'_1 &= a_{11} * b_1 + a_{12} * b_2 + a_{13} * b'_2 + a_{14} * b_3 + a_{15} * b_4 \\ &= p_1 - a_{13} * b_3 - a_{14} * b_4 + a_{13} * b'_2 + a_{14} * b_3 + a_{15} * b_4 \end{aligned}$$

$$\begin{aligned} p'_2 &= a_{21} * b_1 + a_{22} * b_2 + a_{23} * b'_2 + a_{24} * b_3 + a_{25} * b_4 \\ &= p_2 - a_{23} * b_3 - a_{24} * b_4 + a_{23} * b'_2 + a_{24} * b_3 + a_{25} * b_4 \end{aligned}$$

Delete a data symbol. Similar to inserting a data symbol, to update the parity when deleting the i-th symbol from $\mathbf{L}$, we need to retrieve *the smaller between either all the data symbols, or all the parity symbols and the data symbols after position* i.

2.3 Robust Dynamic Provable Data Possession

In this section, we present R-DPDP, Robust Dynamic Provable Data Possession, a new framework to add robustness to dynamic RDC setting. R-DPDP allows to audit remote data that is dynamically changing and, at the same time, offers protection against both large and small data corruption. To the best of our knowledge, robustness has not been previously considered for dynamic remote data checking while maintaining small, constant, client storage. For the convenience of presentation, we build our R-DPDP on top of one specific dynamic RDC scheme, namely, DPDP (Section 2.1.1). The proposed R-DPDP can be easily adapted to other dynamic RDC schemes [13, 14].

We start by summarizing the challenges that need to be overcome when adding robustness to DPDP. We then present the definition of an R-DPDP scheme and propose two R-DPDP constructions: πR-D (an extension of the πR scheme presented in Section 2.1.2)

• $\mathtt{D}$ is the encoded file, $\mathtt{F}$ is the original file, $\mathtt{P}$ is the redundancy added after applying a RS code over $\mathtt{F}$. We have $\mathtt{D} = \mathtt{F}||\mathtt{P}$.
• n is the number of symbols in a constraint group: $n = k + d$, where k is the number of data symbols and d is the number of parity symbols. A (n, k) RS code is applied over each constraint group.
• M is the server metadata computed over D (stored at the server, includes the verification tags).
• $\mathtt{F}_i$ is the i-th version of $\mathtt{F}$. We have $\mathtt{D}_i = \mathtt{F}_i||\mathtt{P}_i$ and M_i is the server metadata for $\mathtt{D}_i$.
• M_c is the client metadata (*e.g.*, the root of the skip list/RSA tree [12]).
• $info$ is the information about the update operation (*e.g.*, full re-write, delete block i, modify block i, insert a block after block i, etc.).

Figure 2.1 Reference sheet for various notations.

and VLCG (Variable Length Constraint Group, a new construction that improves the communication efficiency of πR-D in the Update phase). To facilitate the exposition, we include a reference sheet with various notations in Figure 2.1.

Challenges. It is challenging to add robustness to DPDP in an adversarial setting and also maintain low bandwidth overhead for updates, because:

- Adding robustness to DPDP requires encoding the data using Reed-Solomon codes. RS codes, as a type of linear codes, provide error correction in a static setting, as they compute redundancy over every portion of the original data. Unfortunately, they are not immediately suitable when the data can be dynamically updated. As shown in Section 2.2, for certain update operations (insert/delete), updating even a small portion of the original data imposes a high communication cost (this holds even under a benign setting, in which the server is trustworthy).
- Robustness applies RS encoding over groups of symbols (*constraint groups*) and it requires to **hide the association between symbols and constraint groups** (*i.e.*, the malicious server should not know which symbols belong to the same constraint group). When dynamic updates are performed over file data, the parity of the affected constraint groups should also be updated, which requires knowledge of the data and the parity symbols in those constraint groups (Section 2.2). However, the client cannot simply retrieve only the symbols of the affected constraint groups, as that would reveal the secret of the corresponding constraint groups and break robustness. Moreover, the client cannot simply update and send back only the parity symbols in the affected constraint groups, as that may allow the malicious server to infer which parity symbols are in the same constraint group by comparing the new parity with the old parity.

File representation. We use two independent logical representation of the file for different purposes:

- For the purpose of file updating (during the Update phase), the file is seen as an ordered collection of blocks. Basically, update operations occur at the block level. This is also the representation used for checking data possession (during the Challenge phase), as each block has one corresponding verification tag.
- For the purpose of encoding for robustness, the file is seen as a collection of symbols, which are grouped into *constraint groups* and each constraint group is encoded independently.

For each file block, there is a corresponding verification tag which needs to be stored at the server. Thus, larger file blocks result in smaller additional server storage overhead due to verification tags. On the other hand, efficient encoding and decoding requires the symbols to be from a small size field. As a result, one file block will usually contain multiple symbols. *Each file update operation which is performed at the block level results into several operations applied to the symbols in that block* (for example, when modifying a data block, all the symbols in that block and the corresponding parity symbols should be modified).

Metric. To measure the communication overhead for data updates, we use as a metric the *update bandwidth factor* α defined as

$$\alpha = \frac{the\ amount\ of\ data\ downloaded\ for\ updating\ one\ file\ block}{the\ total\ amount\ of\ data\ at\ the\ server}$$

2.3.1 R-DPDP Definition

We introduce the definition of a robust dynamic provable data possession (R-DPDP) scheme. Compared to a DPDP scheme, R-DPDP adds robustness, which is reflected in slightly different definitions for the data-updating algorithms. We have also added

an explicit algorithm to decode the data, since data decoding is more challenging when robustness is needed.

Definition 2.3.1. (R-DPDP SCHEME) *A Robust Dynamic Provable Data Possession (*R-DPDP*) scheme is a collection of eight polynomial-time algorithms:*

- KeyGen$(1^{\kappa}) \rightarrow \{sk, pk\}$*: a probabilistic key generation algorithm run by the* ***client*** *to setup the scheme.* Input: *the security parameter* κ*.* Output: *the secret key* sk *and public key* pk*.*
- PrepareUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c) \rightarrow \{e(\Delta D), e(info'), e(\Delta M)\}$*: an algorithm run by the* ***client*** *to prepare (a part of) the file for untrusted storage.* Input: *the secret key* sk*, the public key* pk*, (a part of) the file* ΔF*, (a part of) the previous version of the encoded file* ΔD_{i-1}*, information about the update operation* $info$*, and the client metadata* M_c*.* Output: *the "encoded" version of the update data* $e(\Delta D)$ *(add to* ΔD *randomness, sentinels, or simply let* $e(\Delta(D)) = \Delta D$*.* ΔD *is the data to be updated), the "encoded" version of the update information* $e(info')$ *(*$info$ *will be changed to* $info'$*, since updating* ΔF *may lead to updating of the redundancy.* $info'$ *should be changed to fit the encoded version of* ΔD*), and the new server metadata* $e(\Delta M)$*. The client will send* $e(\Delta D), e(info')$*, and* $e(\Delta M)$ *to the server.*
- PerformUpdate$(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M)) \rightarrow \{D_i, M_i, M_c^{'}, P_{M_c'}\}$*: an algorithm run by the* ***server*** *in response to an update request from the client.* Input: *public key* pk*, the old version of the encoded file* D_{i-1}*, the metadata* M_{i-1}*, and the values* $e(\Delta D), e(info)$*,* $e(\Delta M)$ *provided by the client.* Output: *the new version of the encoded file* D_i *and metadata* M_i*, the metadata to be sent to client* $M_c^{'}$ *and its proof of correctness* $P_{M_c'}$*. The server will send* $M_c^{'}$ *and* $P_{M_c'}$ *back to the client.*
- VerifyUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c^{'}, P_{M_c'}) \rightarrow \{accept, reject\}$*: an algorithm run by the* ***client*** *to verify the server's behavior during the update.* Input: *all the inputs from* PrepareUpdate*,* $M_c^{'}$ *and the proof* $P_{M_c'}$ *which are sent back by the server.* Output: accept *if the check succeeds,* reject *otherwise.*
- GenChallenge$(sk, pk, M_c) \rightarrow \{c\}$*: a probabilistic algorithm run by the* ***client*** *to issue a challenge for the server.* Input: *the secret key* sk*, public key* pk*, and the latest client metadata* M_c*.* Output: *the challenge* c *that will be sent to the server.*
- Prove$(pk, D_i, M_i, c) \rightarrow \{\Pi\}$*: an algorithm run by the* ***server*** *to generate the proof of possession upon receiving the challenge from the client.* Input: *the public key* pk*, the latest version of the encoded file* D_i*, the metadata* M_i*, and the challenge* c*.* Output: *a proof of possession* Π *that will be sent back to the client.*
- Verify$(sk, pk, M_c, c, \Pi) \rightarrow \{\text{accept}, \text{reject}\}$*: an algorithm run by the* ***client*** *to validate a proof of possession upon receiving the proof* Π *from the server.* Input: *the secret key*

sk, the public key pk, the client metadata M_c, the challenge c, and the proof Π. Output: accept *if* Π *is a valid proof of possession,* reject *otherwise.*

- Decode$(sk, pk, D_i, M_i, M_c) \rightarrow \{F_i, failure\}$*: an algorithm run by the* ***client*** *to decode the latest version of the encoded file D_i (repair it if small corruption exists).* Input: *the secret key sk, the public key pk, the latest version of the encoded file D_i (where $D_i = F_i||P_i$), metadata M_i, and client metadata M_c.* Output: *the latest version of the file F_i if the decode process is successful,* failure *otherwise.*

A R-DPDP protocol can be constructed in four phases, Setup, Challenge, Update, *and* Retrieve.

Setup*: The client C who is in possession of file F runs $(pk, sk) \leftarrow$ KeyGen(1^κ), followed by $\{e(\Delta D), e(info'), e(\Delta M)\} \leftarrow$ PrepareUpdate$(sk, pk, F, NULL,$ "full re-write", $NULL)$. C sends $e(\Delta D), e(info'), e(\Delta M)$ to the server S. S runs $\{D_1, M_1, M_c^{'}, P_{M_c^{'}}\} \leftarrow$ PerformUpdate$(pk, NULL, NULL, e(\Delta D), e(info'), e(\Delta M))$ and sends $M_c^{'}, P_{M_c^{'}}$ back to C. C then runs* VerifyUpdate$(sk, pk, F, NULL,$ "full re-write", $NULL, M_c^{'}, P_{M_c^{'}})$ *to check whether the initial data outsourcing is successful or not. If successful, C sets $M_c = M_c^{'}$, and deletes F.*

Challenge*: C generates challenge c by running* GenChallenge(sk, pk, M_c)*, and sends c to S. S runs $\{\Pi\} \leftarrow$ Prove(pk, D_i, M_i, c) and sends to* **C** *the proof of possession Π. C can check the validity of the proof Π by running* Verify(sk, pk, M_c, c, Π).

Update*: C downloads ΔD_{i-1} from S, and runs $\{e(\Delta D), e(info'), e(\Delta M)\} \leftarrow$ PrepareUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c)$. C sends $e(\Delta D), e(info'), e(\Delta M)$ to S. S runs $\{D_i, M_i, M_c^{'}, P_{M_c^{'}}\} \leftarrow$ PerformUpdate$(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info'), e(\Delta M))$ and sends $M_c^{'}, P_{M_c^{'}}$ back to C. C then runs* VerifyUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c^{'}, P_{M_c^{'}})$ *to check whether the update is successful or not. If successful, C sets $M_c = M_c^{'}$, then deletes ΔF and ΔD_{i-1}.*

Retrieve*: C downloads the current version of the encoded file D_i and the server metadata M_i, then runs $\{F_i, failure\} \leftarrow$ Decode(sk, pk, D_i, M_i, M_c).*

2.3.2 Enhancing πR: πR-D

We first describe πR-D, an R-DPDP construction obtained by adapting the πR scheme (Section 2.1.2) to add robustness on top of a DPDP scheme DPDP = (KeyGen_DPDP, PrepareUpdate_DPDP, PerformUpdate_DPDP, VerifyUpdate_DPDP, GenChallenge_DPDP,

Prove_DPDP, Verify_DPDP) (refer to [12] for the detailed definition of a DPDP scheme). In the Setup phase, the file $\mathtt{F}$ is first processed according to πR, and the parity data $\mathtt{P}$ is generated. The encoded file $\mathtt{D} = \mathtt{F}||\mathtt{P}$ is further processed using the DPDP algorithms PrepareUpdate_DPDP, PerformUpdate_DPDP, and VerifyUpdate_DPDP, after which the initial file version has been sent to store in the server. During the Challenge phase, we can use directly the DPDP algorithms GenChallenge_DPDP, Prove_DPDP, and Verify_DPDP to verify the integrity of the latest file version.

In the Update phase, the main operations are:

- *Insert/Delete a data block.* A data block may contain multiple symbols, which may belong to more than one constraint groups. Inserting/deleting a data block is equivalent to inserting/deleting all the symbols in that block. Inserting/deleting a data symbol will affect the indices of the following data symbols in the whole file, as well as the parameter f of the PRP ψ in πR. Since in πR the contents of each constraint group are decided based on the indices provided by the PRP ψ, the changing of the parameter f of PRP ψ will require the client to download the entire file $\mathtt{F}$ and re-compute the parity $\mathtt{P}$ based on a new set of constraint groups. The update bandwidth factor is $\alpha = \frac{|\mathtt{F}|}{|\mathtt{D}|}$. The updated file $\mathtt{F}$ is pre-processed using the technique described in πR, but the new parity $\mathtt{P}$ will be permuted and encrypted by a new key (the client will only keep the new key and discard the previous key). For an insert operation, the newly inserted block is sent back to the server using the corresponding DPDP update algorithms, whereas for a delete operation the corresponding block should be deleted. Also, $\mathtt{P}$ should replace the old parity at the server.
- *Modify a data block.* The client downloads the data block to be modified (*i.e.*, the old data block) and the latest version of the parity $\mathtt{P}$, decrypts $\mathtt{P}$ and restores the original order, updates the parity symbols in the affected constraint groups according to $\mathtt{P}$, the data symbols in the old and the new data block (exact procedure described in Section 2.2). The update bandwidth factor is $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$. To prevent the server from learning the contents of the constraint groups by comparing the new parity with the old parity, the client should use a new key to permute and encrypt the parity symbols (the client will keep the new key and discard the previous key). The new data block and the new parity $\mathtt{P}$ are sent back to the server using the DPDP update algorithms, replacing the corresponding old block and old $\mathtt{P}$.

In the Retrieve phase, the client simply retrieves the file $\mathtt{F}$ and may use the parity $\mathtt{P}$ to correct data corruption.

Table 2.1 Statistics for Update Operations Based on Two CVS Repositories of OpenSSL and Eclipse

	OpenSSL	Eclipse
dates of activity	1998-2011	2001-2011
# of files	4,283	180,662
# of commits	67,846	883,045
# of insertions (lines)	707,978	8,579,577
# of deletions (lines)	678,936	7,009,582
# of modifications (lines)	371,159	6,714,823
Avg. # commits/file	15.8	4.9
Avg. # insertions/commit	10.4	9.7
Avg. # deletions/commit	10	7.9
Avg. # modifies./commit	5.5	7.6

Performance analysis. πR-D is efficient during Setup and Retrieve, but has high communication overhead during Update, since for every insertion/deletion the update bandwidth factor is $\alpha = \frac{|\mathtt{F}|}{|\mathtt{D}|}$, which approaches 1 in practice. We have analyzed the pattern of updates for the source files of two popular projects, OpenSSL [44] and Eclipse [45]. Table 2.1 shows that the number of insert and delete operations, compared to modifications, represent a majority of the total number of updates. Thus, it is likely that one small update may require to download the entire file $\mathtt{F}$. We need a construction with lower communication overhead for data updates.

Security analysis. The δ-robustness (Section 1.3) of πR-D can be established similarly as for πR [8]: Once we fix a target for the probability of a successful attack (*e.g.*, 10^{-10}), we can determine the RS encoding parameters that minimize the number of blocks being spot-checked during a challenge. The resulting RS encoding provides the value of δ. If the adversary corrupts more than a δ-fraction of the encoded file, the spot-checking based strategy and the authentication structure of the underlying DPDP scheme guarantee that the auditor can detect such corruptions with high probability. If the adversary corrupts at most a δ-fraction, the data can be retrieved based on the recovery capability of the RS code.

2.3.3 Variable Length Constraint Group

Though efficient in encoding, πR-D has a high communication overhead for updates. In πR-D, the PRP ψ is applied to the index of data symbols, thus making it sensitive to insert/delete operations (*e.g.*, one simple insertion/deletion may require the client to download and to re-encode the entire file $\mathtt{F}$).

To mitigate the drawbacks of πR-D, we propose a second construction called Variable Length Constraint Group (VLCG). Like in πR-D, we still use the notion of *constraint groups*, which are groups of symbols over which an RS code is computed. However, we rely on two additional main insights.

Firstly, unlike in πR-D, in which symbols are assigned to constrained groups based on the position (*i.e.*, index) of the symbols in the file, VLCG assigns symbols to constraint groups based on the value of the symbols. More precisely, for a data symbol b, we use $h_K(b)$ to decide the index of the constraint group to which b belongs. This has the advantage that, after we have inserted/deleted a data symbol into/from f, to update the parity, we can insert/delete the data symbol into/from the corresponding constraint group, *without affecting other constraint groups*.

Secondly, we employ several techniques to preserve robustness and minimize the bandwidth overhead. For example, we reduce insert operations to append operations and delete operations to modify operations when updating the RS-coded parity data.

We seek to maintain *the same fault tolerance level* (see definition in Section 2.2) for all constraint groups (that is, for every (n, k) constraint group, $d = n - k$ will be kept the same after each update operation). All the parity symbols $\mathtt{P}$ should be permuted like in πR, but there is no need to encrypt them (this is explained later in more detail).

Next, we give an overview of the VLCG construction, focusing on the Update and Retrieve phases.

Update operations. For all update operations, we first execute the actual block update on the file data, but the challenging step is how to efficiently update the RS-coded parity data.

Inserting a symbol into the file requires updating the parity symbols of the constraint group to which the symbol is assigned. According to the analysis in Section 2.2, inserting a symbol into a RS code (equivalent to inserting it into a constraint group) requires to retrieve either all the data symbols in that constraint group, or all the parity symbols and some of the data symbols in that constraint group. As we argued for the πR scheme (Section 2.1.2), to ensure the δ-robustness guarantee, it is necessary that the association between symbols and constraint groups remains hidden. Thus, it is insufficient to only retrieve symbols from the corresponding constraint group. Moreover, it is not possible for the client to efficiently determine which other symbols belong to that constraint group considering that the whole file has been outsourced to the server and then deleted from the client. For these reasons, the client would have to retrieve the entire file $\mathtt{F}$. We overcome this limitation by: After we have inserted a data symbol to the file, to update the parity symbols in the corresponding constraint group, we always append this symbol to the end of the data symbols in that constraint group (note that this operation is only for the purpose of updating parity, of course, the symbol is physically inserted in the file at the desired location). The advantage of this method is that appending a data symbol to a Cauchy RS code does not require to download any data symbols and we can update the corresponding parity symbols based only on the old parity (cf. Section 2.2). We note that ensuring δ-robustness prevents us from retrieving only the parity symbols from this constraint group. Instead, we retrieve all

the parity data $\mathtt{P}$ (refer to Section 2.5 for further optimization). Thus the update bandwidth factor is $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

Deleting a data symbol is more complex. Although under a benign setting this operation could be achieved by only retrieving symbols from the same constraint group, ensuring δ-robustness prevents us from using this strategy. Instead, we use a different strategy: To delete a data symbol, we ask the server to physically delete the symbol from $\mathtt{F}$, but we update the parity symbols from the corresponding constraint group *as if that symbol was modified to have the value 0* (note that this operation is only for the purpose of updating parity, of course, the symbol is physically deleted from the file). As a result, the delete operation is converted into a modify operation when updating the RS-encoded parity data for the corresponding constraint group (a similar strategy was previously used in [11, 28]). A code modify operation, according to Section 2.2, only requires to download the parity data and the old symbol from the corresponding constraint group. This means that the update bandwidth factor can be kept as $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$ for deletion.

Modifying a data symbol is the most complex operation because if a symbol is modified to a new different value, it may be re-assigned to a different constraint group. The old symbol must first be deleted from its current constraint group and the new symbol must be inserted into a (possibly) new constraint group (*i.e.*, it is a combination of an insertion and a deletion). The update bandwidth factor remains $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

In Section 2.5, we propose an alternative method to optimize the communication overhead for updates ($\alpha = \frac{\log^2 |\mathtt{P}|}{|\mathtt{D}|}$) by using Oblivious RAMs to only retrieve the parity symbols from the corresponding constraint groups.

Retrieve data. The method we use for deciding to which constraint group does a symbol belong in VLCG introduces an additional challenge in the Retrieve phase. By using a PRF over the value of the symbol to decide its constraint group, the encoded file contains no information about the relative position of the symbols inside a constraint group. Note that the initial position of the symbols inside a constraint group may change because of update operations (*i.e.*, modification of symbols). In case of data corruption, the RS code computed over a constraint group will be used to recover the original symbols; however, successfully decoding the RS code requires knowledge of the correct position of symbols inside the constraint group. During file recovery, the correct position of symbols inside a constraint group may be uncertain because of two reasons: (a) if a symbol is corrupted, the client does not know to which constraint group did that symbol belong, and thus the symbol will be missing from that constraint group during RS decoding; (b) if a symbol is deleted (*i.e.*, a valid delete operation), it does not exist anymore at the server (*i.e.*, we cannot find this symbol in the latest file version), but for RS decoding purposes the client should still use a symbol with value 0 at the corresponding position in the constraint group to which the symbol belongs (recall previously when deleting a symbol, we update the parity symbols from the corresponding constraint group as if that symbol was modified to have the value 0).

We illustrate the uncertainty in decoding with an example. Assume that a constraint group is a $(6, 4)$ RS code $(b_1\ b_2\ b_3\ b_4\ p_1\ p_2)$. If b_3 was corrupted, the RS decoding should take as input $(b_1\ b_2\ ?\ b_4\ p_1\ p_2)$, whereas if b_3 was deleted, the input for RS decoding should be $(b_1\ b_2\ 0\ b_4\ p_1\ p_2)$. But how does the client know that in position 3 there should be a corrupted symbol or a 0 symbol?

To deal with the uncertainty about symbol position in a constraint group during decoding, we propose a strategy similar with the one used in HAIL [21]: To identify the correct locations of data symbols (healthy and 0 symbols) in their corresponding constraint groups, we convert the parity symbols into cryptographically secure Message Authentication Codes (MACs). Based on these MACs, we use a brute force approach to determine the correct position of symbols for RS decoding (full details in Figure 2.2). The parity symbols are converted to secure MACs by composing them with a pseudorandom function (PRF) (we call this operation *masking*). The PRF is computed over the file identifier, the index of the corresponding constraint group, and the index of the parity symbol in the constraint group. For example, using the (6,4) RS code described in Section 2.2, the new parity symbols p'_1 and p'_2, which are also secure MACs over b_1, b_2, b_3, b_4, are computed as:

$p'_1 = p_1 + g_{K'}(file\_id||constraint\_group\_index||5)$

$p'_2 = p_2 + g_{K'}(file\_id||constraint\_group\_index||6)$,

where g is a PRF and all operations are over $\mathbb{F}_{2^w}$, in which "$+$" and "$-$" can be regarded as bitwise XORs. To strip off g from p'_1 and p'_2, we compute:

$p_1 = p'_1 - g_{K'}(file\_id||constraint\_group\_index||5)$

$p_2 = p'_2 - g_{K'}(file\_id||constraint\_group\_index||6)$

Since different constraint groups may end up having different sizes, the client needs to keep track of the size of each constraint group [2]. This can be done by recording either n or k for each (n, k) RS code (because $d = n - k$ is fixed for all constraint groups). Let Φ be the set of (n, k) parameters for all constraint groups. For ease of presentation, we assume

[2]Knowledge of the (n, k) parameters is required for the operations in the Update and Retrieve phases.

that Φ is stored (and updated) at the client. In Section 2.5 we discuss how to store Φ more efficiently.

Finally, we note that since the parity symbols are masked with a PRF, there is no need to further encrypt them as in step 3 of the πR construction (described in Section 2.1.2).

The VLCG construction. We are now ready to present the details of our main R-DPDP construction, Variable Length Constraint Group (VLCG). We fix the parameters n and k as the initial size of the RS code computed over each constraint group of k data symbols and let $d = n - k$ be the fault tolerance level of the code. For a file with f symbols $\mathtt{F} = \{\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_f\}$, there will be $m = f/k$ constraint groups and each constraint group will initially have approximately n symbols (k data and d parity symbols). As file updates are performed, the (n, k) parameters for different constraint groups will change, but the fault tolerance level $d = n - k$ will be preserved. Each file symbol is an element in the Galois Field $\mathbb{F}_{2^w}$.

Let κ be a security parameter. In addition, we make use of a pseudo-random permutation (PRP) φ and two pseudo-random functions (PRF) h and g with the following parameters:

- $\varphi : \{0,1\}^\kappa \times \{0,1\}^{md} \rightarrow \{0,1\}^{md}$
- $h : \{0,1\}^\kappa \times \{0,1\}^w \rightarrow \{0,1\}^{\log m}$
- $g : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^w$

Figures 2.2 and 2.3 describe our VLCG construction, which can be built on top of any DPDP scheme DPDP = (KeyGen_DPDP, PrepareUpdate_DPDP, PerformUpdate_DPDP, VerifyUpdate_DPDP, GenChallenge_DPDP, Prove_DPDP, Verify_DPDP). We construct VLCG in four phases Setup, Challenge, Update, and Retrieve as follows.

KeyGen(1^κ):

1. $\{sk_{DPDP}, pk\} \leftarrow$ KeyGen_DPDP(1^κ), and $K, K' \xleftarrow{R} \{0,1\}^\kappa$
2. Return $\{sk = \{sk_{DPDP}, K, K'\}, pk\}$

PrepareUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c)$:

1. If $info$ ="$full\ re\text{-}write$ " /*occurs in the Setup phase, to prepare the original file for outsourcing. ΔF is the original version of the file, $i = 1$*/
   - P = ComputeParityData$(sk, pk, \Delta F, NULL, 0)$, $\Delta D = \Delta F || P$, $info'$ = "$full\ re\text{-}write$"
   - Return PrepareUpdate_DPDP$(sk_{DPDP}, pk, \Delta D, info', M_c)$

   /*For ease of presentation, we only consider block-level updates (this can be easily extended to arbitrary file portions), thus, in the following, the block $B = \Delta F$. If $info$ is "delete" or "modify" a block, then ΔD_{i-1} is $P_{i-1}||B'$, and B' denotes the block to be deleted or modified. If $info$ is "insert" a block, then ΔD_{i-1} is P_{i-1}.*/
2. Restore the original order of symbols in P_{i-1} and strip off PRF g from them
3. If $info$ = "$insert\ B$"
   - P_i = ComputeParityData$(sk, pk, B, P_{i-1}, 1)$, $\Delta D = P_i||B$, $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ insert\ block\ B$"
4. Else if $info$ = "$delete\ B'$ "
   - P_i = ComputeParityData$(sk, pk, B', P_{i-1}, 2)$, $\Delta D = P_i$, $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ delete\ block\ B'$"
5. Else if $info$ = "$modify\ B' to\ B$"
   - P_i = ComputeParityData$(sk, pk, B'||B, P_{i-1}, 3)$, $\Delta D = P_i||B$, $info'$ = "$replace\ P_{i-1}\ with\ P_i,\ modify\ B'\ to\ B$ "
6. Return PrepareUpdate_DPDP$(sk_{DPDP}, pk, \Delta D, info', M_c)$

PerformUpdate$(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M))$:
Return PerformUpdate_DPDP$(pk, D_{i-1}, M_{i-1}, e(\Delta D), e(info), e(\Delta M))$

VerifyUpdate$(sk, pk, \Delta F, \Delta D_{i-1}, info, M_c, M_c', P_{M_c'})$:

1. Re-compute ΔD and $info'$ according to the procedure in PrepareUpdate /*In fact, ΔD and $info'$ can directly be stored at the client after PrepareUpdate, and there is no need to re-compute them*/
2. Return VerifyUpdate_DPDP$(sk_{DPDP}, pk, \Delta D, info', M_c, M_c', P_{M_c'})$

GenChallenge(sk, pk, M_c): Return GenChallenge_DPDP(sk_{DPDP}, pk, M_c)

Prove(pk, D_i, M_i, c): Return Prove_DPDP(pk, D_i, M_i, c)

Verify(sk, pk, M_c, c, Π): Return Verify_DPDP$(sk_{DPDP}, pk, M_c, c, \Pi)$

Decode$(sk, pk, D_i = F_i||P_i, M_i, M_c)$:

1. Check the freshness of the retrieved file using the dynamic verification structure (*e.g.*, for DPDP that uses a skip list, re-compute the root of the skip list using the verification tags as leaves and compare to the root stored at the client, which is part of M_c [12]). If the check fails, then return $failure$.
2. Check the data blocks in F_i using the corresponding verification tags in M_i and discard the corrupted blocks. If there are no corrupted data blocks, return F_i. Otherwise, assign the symbols in healthy data blocks to their constraint groups (*i.e.*, for a symbol b, the index of its constraint group is $h_K(b)$).
3. Re-order all the parity symbols in P_i, strip off PRF g from them, and put them back to their right locations in the corresponding constraint groups based on the (n, k) parameters of each constraint group.
4. For each (n, k) constraint group, apply brute force decoding as follows. Let k' be the number of healthy symbols that have been assigned to this constraint group. Consider all permutations of k' symbols in k' locations (out of k locations for data symbols), together with $k - k'$ parity symbols (out of d parity symbols): (a) apply RS decoding on the k symbols to recover the original data symbols; (b) re-compute the parity symbols; (c) if at least one of the newly computed parity symbols match the parity symbols retrieved from the server, then the decoding for this constraint group is considered successful. If there are no successful decodings, then further consider ℓ of the $k - k'$ locations as zero symbols (with $1 \leq \ell \leq k - k'$), together with $k - k' - \ell$ parity symbols (out of d parity symbols), and further decode the k symbols as previously described. If there are still no successful decodings, then return $failure$.
5. Return the successfully decoded file F_i.

Figure 2.2 VLCG: an R-DPDP construction.

ComputeParityData$(sk, pk, \mathcal{B}, P_{i-1}, flag)$:
(run by the client to compute the parity in Setup phase or update the parity in Update phase)

1. If $flag$ = 0 /*compute the parity for the original file in Setup phase. $\mathcal{B}$ represents the original file*/
   - For each symbol b in file $\mathcal{B}$, compute $h_K(b)$ to determine to which constraint group will b be assigned
   - For each constraint group with k symbols (k may be different for different groups), apply a $(k+d, k)$ RS code
   - Let P_i be the collection of all the md parity symbols
   - All the parity symbols in P_i are masked using PRF g and permuted using PRP φ, both keyed with K'
   - Return P_i
2. Else if $flag$ = 1 /*insertion, $\mathcal{B}$ represents the block to be inserted*/
   - For each symbol b in block $\mathcal{B}$, update in P_{i-1} the constraint group with index $h_K(b)$ by appending b to the data symbols of this constraint group
   - P_i = P_{i-1}
3. Else if $flag$ = 2 /*deletion, $\mathcal{B}$ represents the block to be deleted*/
   - For each symbol b in block $\mathcal{B}$, update in P_{i-1} the constraint group with index $h_K(b)$ by modifying b to the zero symbol
   - P_i = P_{i-1}
4. Else if $flag$ = 3 /*modification of B' to B, $\mathcal{B} = B'||B$*/
   - For each symbol b in block B', update in P_{i-1} the constraint group with index $h_K(b)$ by setting b to be the zero symbol
   - For each symbol b in block B, update in P_{i-1} the constraint group with index $h_K(b)$ by appending b to the data symbols of this constraint group
   - P_i = P_{i-1}
5. All the parity symbols in P_i are masked using PRF g and permuted using PRP φ, both with a new key K' (the previous K' will be discarded after having verified the update successfully)
6. Return P_i

Figure 2.3 Computing the parity symbols in VLCG.

Setup. The client C runs $\{sk, pk\} \leftarrow \mathsf{KeyGen}(1^\kappa)$, and then runs PrepareUpdate on the file F. PrepareUpdate applies the PRF h over every symbol in F, determining the group where each symbol is assigned to (there are m groups). For every group of k symbols (k may be different for different groups, depending on how many symbols are assigned to the groups), PrepareUpdate applies a $(n = k+d, k)$ RS code and every group becomes a (n, k) constraint group. The md parity symbols from all the constraint groups will form the parity data P. All the symbols in P are masked using PRF g and permuted using PRP φ, both keyed with key K' (similar as in πR). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process $\mathtt{D} = \mathtt{F}||\mathtt{P}$.

The output of PrepareUpdate is sent to the server S. S runs PerformUpdate to fully re-write the data (D and the corresponding verification tags in M) and sends back the proof. C verifies the proof by running VerifyUpdate. If the verification is successful, C discards

D and M, and keeps all the (n, k) parameters; otherwise, C quits the protocol and retries with a different server.

Challenge. As described in Section 2.3.1, the client challenges the server to prove data possession using the GenChallenge, Prove, and Verify algorithms which simply call their counterpart algorithms of DPDP.

Update. Three operations are available in the Update phase: insert, delete, and modify a data block.

- *Insert a data block B.* After having downloaded the whole file parity $\mathtt{P}_{i-1}$ (*i.e.*, $\Delta D_{i-1} = \mathtt{P}_{i-1}$), C runs PrepareUpdate. C first restores the original order of the parity symbols in $\mathtt{P}_{i-1}$ and then strips off PRF g from them. For each symbol $\mathtt{b}$ in B, C updates the constraint group with index $h_K(\mathtt{b})$ by appending $\mathtt{b}$ to the data symbols of the constraint group (cf. Section 2.2). C obtains $\mathtt{P}_i$ by using PRF g to mask all the symbols in the file parity P and by using PRP φ to permute them (g and φ are keyed with a new key K'). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process the data $\mathtt{P}_i||B$. The output of PrepareUpdate is sent to S. S runs PerformUpdate and sends back the proof for updating the corresponding data correctly. C then runs VerifyUpdate to check the proof. If successful, C discards $\mathtt{P}_{i-1}$, B, and the old key K', and updates the (n, k) parameters for the corresponding constraint groups to $(n + 1, k + 1)$; otherwise, C aborts the protocol and raises an alarm.
- *Delete a data block B'.* After having downloaded the whole parity data $\mathtt{P}_{i-1}$ and the block B' that is to be deleted (*i.e.*, $\Delta D_{i-1} = \mathtt{P}_{i-1}||B'$), C runs PrepareUpdate. It first restores the original order of the parity symbols in $\mathtt{P}_{i-1}$ and strips off PRF g from them. For each symbol $\mathtt{b}$ in B', it updates the constraint group with index $h_K(\mathtt{b})$ by modifying the value of $\mathtt{b}$ to be zero (cf. Section 2.2). C obtains $\mathtt{P}_i$ by using PRF g to mask all the symbols in the file parity P and by using PRP φ to permute them (g and φ are keyed with a new key K'). PrepareUpdate then calls the PrepareUpdate_DPDP algorithm of DPDP to further process $\mathtt{P}_i$. The output of PrepareUpdate is sent to S. PerformUpdate and VerifyUpdate are run just like in the insert a block operation above, except that there is no need to update the (n, k) parameters after C verifies the update successfully.
- *Modify a data block B' to B.* Modifying a data block B' to B is equivalent to first deleting the old block B' and then inserting the new block B.

For each of these operations (insert, delete, modify a block), the client discards the previous key K' after running VerifyUpdate and replaces it with the new K'. Thus, C only stores a constant amount of key material.

Retrieve. C downloads D_i (the latest version of the encoded file D) and metadata M_i. C then applies the Decode algorithm. If Decode returns $failure$, C should raise an alarm.

The Decode algorithm relies on brute force decoding to recover the file (VLCG can tolerate up to $d-1$ erasures in each constraint group). We argue this is not a major concern because, firstly, file recovery is usually a rare event; secondly, in Section 2.5 we present an optimization that trades-off client computation during decoding for additional server storage. We also note that during Decoding, verification tags are used to determine the healthy data blocks and only symbols in healthy data blocks are assigned to their constraint groups (thus, if the server swaps two symbols in the data file, the corresponding blocks will be considered corrupted).

2.4 Security and Performance Analysis for VLCG

Security analysis for VLCG. The main difference between VLCG and πR-D in the Setup phase is that VLCG determines the constraint groups by applying a PRF over the content of data symbols, whereas πR-D determines them by applying a PRF over the indices of the data symbols. The security properties of the PRF used to decide the constraint groups and of the randomized encryption applied on the file data before being stored at the server, ensure that the server can infer which symbols are in the same constraint group with negligible probability. Moreover, VLCG masks and permutes the parity in a similar fashion

with the permuting and encrypting of the parity in πR-D, ensuring a similar δ-robustness guarantee as for πR-D (see security analysis for πR-D).

The security of the Challenge and Update phases in VLCG relies on the security of the underlying DPDP scheme (which ensures that large corruptions will be detected based on spot-checking and that the server possesses the latest version of the file based on the authenticated structure).

In the Update phase, for insert/delete operations, VLCG only downloads the parity, updates the corresponding parity symbols, then re-masks and re-permutes the parity with a new key, which is comparative to πR-D, in which although the whole file data is downloaded (for the purpose of re-computing the parity), only the newly generated parity is re-permuted and re-encrypted with a new key; for modify operations, both VLCG and πR-D only download the parity (rather than the whole file data) and update the parity correspondingly. In the Retrieve phase, both schemes rely on the same mechanisms (metadata and authenticated data structure) to detect corruption. This implies that VLCG and πR-D provide a similar security level. We leave a rigorous security analysis of VLCG as future work.

Performance analysis for VLCG. For every update operation (insertion, deletion, and modification), the update bandwidth factor α for VLCG is approximately $\frac{|\mathtt{P}|}{|\mathtt{D}|}$, whereas for πR-D α is approximately $\frac{|\mathtt{F}|}{|\mathtt{D}|}$ for insertion/deletion. Since d is usually small compared to k (*i.e.*, $|\mathtt{P}|$ is a lot smaller than $|\mathtt{F}|$), the communication overhead for updates in VLCG is significantly smaller than in πR-D.

The encoding computation for VLCG is close to that of πR-D, as expressed in the following theorem:

Theorem 2.4.1. *Let f be the number of symbols in the file F. If $C_1(f)$ is the computation for encoding in πR-D, and $C_2(f)$ is the computation for encoding in VLCG, then we have:* $C_2(f) = \Theta(C_1(f))$.

Proof. In πR-D, the encoding computation of one constraint group is approximately $c(\frac{f}{m})^2$ (the computation for Cauchy RS encoding is approximately quadratic. d is small compared to $\frac{f}{m}$), while c is constant, thus, $C_1(f) \approx m * c(\frac{f}{m})^2 = c\frac{f^2}{m}$.

In VLCG, the f symbols are distributed to m groups by using PRF h. Assume the number of data symbols in the m constraint groups are k_1, k_2, ..., k_m. Then, $C_2(f) \approx c(k_1^2 + k_2^2 + ... + k_m^2)$.

Let X be a random variable that denotes the number of data symbols in one constraint group. We have:

The expected value of X: $E(X) = \frac{f}{m}$

The variance of X: $Var(X) = E(X^2) - (E(X))^2$

$$\begin{aligned} C_2(f) &\approx c(k_1^2 + k_2^2 + ... + k_m^2) \approx c(mE(X^2)) \\ &= cm(E(X)^2 + Var(X)) = cm((\frac{f}{m})^2 + Var(X)) \\ &= c(\frac{f^2}{m} + mVar(X)) \end{aligned}$$

$\lim_{f\to\infty}(\frac{C_2(f)}{C_1(f)}) = \lim_{f\to\infty}(1 + Var(X)\frac{m^2}{f^2}) \approx 1$ (if h is a good PRF and the input for h is random enough, $Var(X)$ will be approximately constant). Thus, $C_2(f) = \Theta C_1(f)$.

□

2.5 Discussion

Unlike πR-D in which all the constraint groups have the same fixed size, in VLCG the length of each constraint group is variable. The client can determine which symbols belong to the same constraint group by keeping track of the set Φ of (n, k) parameters for all the constraint groups. A basic approach is to store Φ at the client, which will result in $O(m)$ additional client storage. An alternative approach is to store Φ at the server and securely manage it using a Merkle hash tree; the client only stores the root of the tree, thus preserving the $O(1)$ client storage overhead; the communication overhead is increased by $O(m)$ during challenges of data possession, and by $O(\log m)$ during updates.

The number k of data symbols in a constraint group should be kept relatively small, so that the (n, k) RS code can be computed efficiently over the constraint group. Once the n and k parameters are fixed, the number of constraint groups m is determined as $m = f/k$. As updates are performed, m remains the same, but the number of data symbols k in each constraint groups may change (though the level of fault tolerance $d = n - k$ is preserved). Since delete operations do not reduce the size of the (n, k) RS code, k will increase over time due to insert/modify operations. To avoid a prohibitive increase of k and keep the (n, k) RS codes efficient to compute, as well as keep the zero symbols as few as possible (to keep the brute force computation in the Decode algorithm at a minimum), the client C should periodically perform a *dynamic adjustment*: After a certain number of updates C retrieves the file and runs Setup to pre-process the file again; in this process, C may pick a different m, depending on the size of the updated file. The amortized bandwidth factor for updates will remain $\alpha = \frac{|\mathtt{P}|}{|\mathtt{D}|}$.

We now describe one step that was previously omitted to simplify the description of the VLCG construction. The assignment of data symbols to constraint groups is done

based on the value of each symbol. To ensure robustness, the server should not know which symbols belong to the same constraint group. The client applies a layer of encryption before storing the file at the server in order to hide that two equal symbols are mapped to the same constraint group. Specifically, in the Setup phase, the succession of steps for the client is: (1) encode the file and obtain the parity, (2) encrypt the file blocks (using randomized encryption), (3) mask and permute the parity, (4) generate verification metadata over the encrypted file and parity, (5) store the encrypted file, parity, and metadata at the server. As a result, the client needs to remove this encryption layer whenever it gets file data from the server, and add the encryption whenever it stores blocks at the server.

Finally, we remark that our VLCG construction is better suited for files with random data in order to ensure that PRF h provides a balanced distribution of symbols into constraint groups. For instance, files could be encrypted before applying the PRF, and then be stored encrypted at the server (this may be desirable anyway if the data is of sensitive nature).

Optimizing the worst-case computation for decoding in VLCG. The Decode algorithm in VLCG has a high worst-case computation, since we do not know the correct positions of healthy data symbols in their corresponding constraint groups. We propose to record the position of each data symbol in its constraint group. For a file $\mathtt{F}$ with f symbols, we maintain $\mathbf{V}$ a vector of positions with f elements. The vector $\mathbf{V}$ is stored encrypted at the server (each element in $\mathbf{V}$ is encrypted independently), which prevents the server from learning information about constraint groups. $\mathbf{V}$ is regarded as part of the file data during the Setup and Challenge phases (thus, there will be verification tags computed over blocks made of elements of $\mathbf{V}$).

In the Update phase, changes on the file symbols should be mirrored by changes in $\mathbf{V}$: Whenever symbols are inserted, deleted, or modified in the file, a corresponding element in $\mathbf{V}$ should be inserted, deleted, or modified, respectively (reflecting the symbol's new position in its constraint group).

In the Retrieve phase, $\mathbf{V}$ is retrieved together with all the data D. After having checked the verification tags on $D||\mathbf{V}$, the elements in $\mathbf{V}$ which have been found corrupted are discarded. The healthy elements in $\mathbf{V}$ will indicate the position of each symbol inside its constraint group. A lightweight brute force computation may still be required because, firstly, for the corrupted elements in $\mathbf{V}$, the client will lose the position information of the corresponding (healthy) data symbols, thus, the positions of these symbols in their constraint groups would have to be determined by brute force; secondly, symbols with 0 value introduced by "delete" and "modify" operations may exist in some constraint groups, and the right positions for such symbols (if needed) will be determined by brute force search. The parameters of the RS code should be selected to ensure that the brute force search during Decode remains reasonable (*e.g.*, for a code with $n = 140$, $k = 128$, $d = 12$, which requires to spot-check 1188 blocks in the Challenge phase to guarantee high probability of corruption detection [8], the worse case running time for brute force is $\binom{140}{12}$, which is approximately 2^{38}). This computation is reasonable considering that data retrieval is a rare event.

Further optimizing the update communication. Compared to πR-D, which requires to download the entire file for every update, VLCG only needs to download the parity symbols. Although the parity is usually very small compared to the whole data, the asymptotic communication per update is $O(f)$, where f is the number of file symbols.

To further optimize the update communication to $O(\log^2 f)$, we can use Oblivious RAM (ORAM) techniques [46]. To update one symbol, instead of downloading the parity $\mathtt{P}$ over the entire file, we use ORAM over $\mathtt{P}$ to only retrieve the d parity symbols of the constraint group corresponding to the updated symbol. These symbols are updated (cf. Section 2.3.3) and then written back to the server using ORAM. For example, when using the ORAM scheme in [47], the amortized communication for every update will be reduced to $O(\log^2 f)$, at the cost of a slight increase in server storage (asymptotically, the server storage remains $O(f)$) and server computation (by $O(\log^2 f)$).

Verification tags in VLCG. Remote data checking schemes designed for the static case [7, 10] embed the index of a block (*i.e.* its position in the file) inside the corresponding verification tag. In order to support efficient dynamic updates, verification tags in DPDP [12] are fundamentally different, as they do not embed the block indices inside the verification tags. In DPDP, the tag for a block B is computed as $g^B \mod N$, where $N = pq$ is a product of two large primes p, q, and g is an element of high order in $\mathbb{Z}_N^*$. Since N should be at least 1024 bits, when the block size is smaller than 1024 bits the size of a tag will be larger than the size of a data block, which may result in an undesirably high additional storage overhead. [13] provides another tag construction which can avoid this issue, but the tags will be of the same size as the data blocks and the verification process is based on expensive bilinear maps. In the following, we provide an alternative to compute the verification tags in a prime-order field and the size of the verification tags will be smaller than the data even when choosing the block size as small as a few hundred bits.

We adopt the tag construction method from [48]. Suppose every file block contains s symbols (*i.e.*, block m_i consists of symbols $m_{i1}, m_{i2}, ..., m_{is}$). Choose a group $\mathbb{G}$ of

prime order p, with $p > max(2^{\lambda}, 2^{w})$ (λ is a security parameter, and w is the length of the symbol). Choose generators $g_i \overset{R}{\leftarrow} \mathbb{G}$ for $i = 1, .., s$. The public key $pk = (p)$, and secret key $sk = (g_1, ..., g_s)$.

In the Setup phase, the tag $\mathcal{T}_{m_i}$ for the block m_i is computed as $\mathcal{T}_{m_i} = \Pi_{k=1}^{s} g_k^{m_{ik}}$.

In the Challenge phase, the client sends to the server pairs (i_j, a_j), for $1 \leq j \leq C$, where C is the number of blocks to be challenged. The server runs Prove_DPDP and returns a proof (T, μ_k), where

$$T = \Pi_{j=1}^{C} \mathcal{T}_{m_{i_j}}^{a_j}, \quad \mu_k = \sum_{j=1}^{C} a_j m_{i_j k}$$

for $1 \leq k \leq s$ ($m_{i_j k}$ is the k-th symbol in block m_{i_j}).

The client runs Verify_DPDP and accepts if $T = \Pi_{k=1}^{s} g_k^{u_k}$ and if M_c verifies.

Proof of correctness:

$$T = \Pi_{j=1}^{C} \mathcal{T}_{m_{i_j}}^{a_j} = \Pi_{j=1}^{C} (\Pi_{k=1}^{s} g_k^{m_{i_j k}})^{a_j} = \Pi_{k=1}^{s} g_k^{\sum_{j=1}^{C} a_j m_{i_j k}}$$

$$= \Pi_{k=1}^{s} g_k^{u_k}$$

CHAPTER 3

REMOTE DATA CHECKING FOR NETWORK CODING-BASED DISTRIBUTED STORAGE SYSTEMS

This chapter introduces RDC-NC, an RDC scheme for Network Coding-based distributed storage systems. RDC-NC is the first RDC protocol built specifically for storage systems which use network codes to distribute data redundantly across multiple untrusted servers. It can be used to ensure data remain intact when faced with data corruption, replay, and pollution attacks.

Remote data checking (RDC) has been shown to be a valuable technique by which a client (acting as a verifier) can efficiently establish that data stored at an untrusted server remains intact over time [7, 9, 10]. This kind of assurance is essential to ensure long-term reliability of data outsourced at data centers or at cloud storage providers. When used with a single server, the most valuable use of remote data checking lies within its prevention capability: The verifier can periodically check data possession at the server and can thus detect data corruption. However, once corruption is detected, the single server setting does not necessarily allow data recovery. In previous chapter (Chapter 2), we present robust dynamic provable data possession scheme, which ensures that data can be recovered upon small corruptions. Unfortunately, it cannot allow data recovery upon large corruptions. Thus, remote data checking has to be complemented with storing the data redundantly at multiple servers. In this way, the verifier can use remote data checking with each server and, upon detecting data corruption (*e.g.*, a large corruption) at any of the servers, it can

use the remaining healthy servers to restore the desired level of redundancy by storing data on a new server.

The main approaches to introduce redundancy in distributed storage systems are through *replication*, *erasure coding*, and more recently through *network coding* [18, 19]. The basic principle of data replication is to store multiple copies of the data at different storage servers, whereas in erasure coding the original data is encoded into fragments which are stored across multiple storage servers. In network coding, the coded blocks stored across servers are computed as linear combinations of the original data blocks.

Network coding for distributed storage systems and application scenarios. Network coding for storage [18, 19] provides unusual performance properties well suited to deep archival stores which are characterized by a *read-rarely* workload. The parameters of network coding make reading data more expensive than data maintenance. Similar with erasure coding, network coding can be used to redundantly encode a file into fragments and store these fragments at n servers so that the file can be recovered (and read) from any k servers. However, network coding provides a *significant advantage* over erasure coding when coded fragments are lost due to server failures and need to be reconstructed in order to maintain the same level of reliability: A new coded fragment can be constructed with optimally minimum communication cost by contacting some of the healthy servers (the repair bandwidth can be made as low as the repaired fragment). This is in sharp contrast with conventional erasure codes, such as Reed-Solomon codes [49] which must rebuild the entire file prior to recovering from data loss. Recent results in network coding for storage have established that the maintenance bandwidth can be reduced by orders of magnitude compared to standard erasure codes.

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

Regulatory storage, data escrow, and deep archival applications present read-rarely workloads that match well the performance properties of network coding. These applications preserve data for future access with few objects being accessed during any period of time. Many of these applications do not require sub-file access; they retrieve files in their entirety. Auditing presents several examples, including keeping business records for seven years in accordance with Sarbanes-Oxley and keeping back tax returns for five years. Only those records that are audited or amended ever need to be accessed, but retaining all data is a legal or regulatory requirement. Medical records are equally relevant. The Johns Hopkins University Medical Image Archive retains all MRI, CAT-scan, and X-ray images collected in the hospitals in a central repository of more than 6 PB. A small fraction of images are ever accessed for historical tracking of patients or to examine outcomes of similar cases. Preservation systems for the storage of old books, manuscripts, data sets also present a read-rarely workload. Furthermore, standards for archival storage [50] represent data as an indivisible package and do not support subfile access. In applications, the size of the data and the infrequency of reads dictate that the performance of storage maintenance, re-encoding to mitigate data loss from device or system failures, dominates the performance requirements of read.

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

In this work, we take a holistic approach and propose novel remote data checking techniques to minimize the combined costs of the prevention and repair components. Previous work on remote data checking has focused exclusively on minimizing the cost of the prevention component (*e.g.*, replication [22] and erasure coding [21, 28] based approaches). However, in the distributed storage settings we consider, the cost of the repair component is significant because over a long period of time servers fail and data needs to be re-distributed on new servers. Our work builds on recent work in coding for distributed storage [18, 19], which leverages network coding to achieve a remarkable reduction in the communication overhead of the repair component compared to an erasure coding-based approach. Unfortunately, this work was proposed for a benign setting. In essence, we seek to preserve in an adversarial setting the minimal communication overhead of the repair component when using network coding. The main challenge towards achieving this goal stems from the very nature of network coding: In the repair phase, the client must ensure the correctness of the coding operations performed by servers, without having access to the

original data. At the same time, the client storage should remain small and constant over time, to conform with the notion of outsourced storage.

The need for remote data checking in distributed storage systems. Read-rarely archival storage also requires introspection and data checking to ensure that data are being preserved and are retrievable. Since data are rarely read, it is inadequate to only check the correctness and integrity of data on retrieval. Storage errors, from device failures, torn writes [51], latent errors [52], and mismanagement may damage data undetectably. Also, storage providers may desire to hide data loss incidents in an attempt to preserve their reputation or to delete data maliciously to reduce expenses [7]. Deep archival applications employ data centers, cloud storage, and peer-to-peer storage systems [53] in which the management of data resides with a third party: Not with the owner of the data. This furthers the need for the data owner to check the preservation status of stored data to audit whether the third party fulfills its obligation to preserve data.

The performance properties of remote data checking protocols, such as PDP [7] and PoR [9], also conform to read-rarely workloads. These protocols allow an auditor to guarantee that data are intact on storage and retrievable using a constant amount of client metadata, constant amount of network traffic, and (most importantly) by reading a constant number of file fragments [7]. Large archival data sets make it prohibitive to read every byte periodically. Remote data checking protocols sample stored data to achieve probabilistic guarantees. When combined with error correcting codes, the guarantees can reach confidence of 10^{-10} for practical parameters [16]. Error correcting codes ensure that small amounts of data corruption do no damage, because the corrupted data may be recovered by the code, and that large amounts of data corruption are easily detected, because they must corrupt many blocks of data to overcome the redundancy.

Table 3.1 Parameters of Various RDC Schemes

	Replication (MR-PDP [22])	Erasure Coding (HAIL [21])	Network Coding (RDC-NC)
Total server storage	$n\|\mathbb{F}\|$	$\frac{n\|\mathbb{F}\|}{k}$	$\frac{2n\|\mathbb{F}\|}{k+1}$
Communication (repair phase)	$\|\mathbb{F}\|$	$\|\mathbb{F}\|$	$\frac{2\|\mathbb{F}\|}{k+1}$
Network overhead factor (repair phase)	1	k	1
Server computation (repair phase)	$O(1)$	$O(1)$	$O(1)$

We assume that RDC-NC uses an MBR code, under the additional constraint to minimize the total server storage, as introduced in Section 3.6. For the repair phase, we describe the costs for the case when one storage server fails.

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

3.0.1 Solution Overview

We propose RDC-NC, Remote Data Checking for Network Coding-based distributed storage systems. Table 3.1 compares RDC-NC with previous RDC schemes. The underlying substrate for adding redundancy in RDC-NC is based on network coding, whereas in previous work it is based on replication and erasure-coding.

To ensure the security of the prevention component, we adapt RDC techniques used in the single server setting [10]. We present a scheme in which only the data owner can check data possession (*i.e.*, it is privately verifiable). However, our scheme can be extended using the techniques in [7, 10] to achieve public verifiability for the prevention phase (*i.e.*, anyone, not just the data owner, can challenge the server to prove data possession).

For the security of the repair component, our solution ensures that the data provided by contributing servers is valid and preserves the amount of desired redundancy in the system. This will ultimately ensure that the original data can be recovered after an arbitrarily many repairs. We identify the *replay* and *pollution* attacks. The proposed RDC schemes successfully mitigate these attacks. To render replay attacks harmless, we use a simple but effective solution: The network coding coefficients are stored encrypted on the server; moreover, the client is the one that chooses the coding coefficients and enforces their use. To prevent pollution attacks, we use an additional repair verification tag, which allows the client to check that a server combines its blocks correctly during the repair phase.

3.1 Related Work

Multiple-server RDCs. Single-server RDC schemes [8–15, 24, 25] focus on detecting corruptions among outsourced data, which cannot allow data recovery upon having found out a corruption (specifically, a large corruption). Multiple-server RDC schemes allow both corruption detection, as well as data recovery. Curtmola et al. [22] proposed an efficient RDC scheme for replication-based distributed storage systems, while Bowers et al. [21] and Wang et al. [28] built RDC schemes for erasure coding-based distributed storage systems. Currently, no RDC scheme is available for network coding-based distributed storage systems, which provide unusual performance properties well suited to read-rarely archival stores.

Replay attacks and pollution attacks in network coding. The *replay attacks* we identify may lead to a reduction in data redundancy, similar to the entropy attacks identified by Jiang et al. [54] in a network setting, in which intermediate nodes forward non-innovative

packets. The solution of Jiang et al. relies on checking if a new coded packet is linearly independent with all previously coded packets. In our distributed storage setting, their solution cannot preserve the minimal communication overhead of the repair component. Pollution attacks always exist when network coding is used to improve the throughput of communication over a network, since the untrusted intermediate servers are responsible to re-encode the data blocks. A line of work on signatures for network coding [48,55] ensures that intermediate nodes perform correctly the encoding operations. Our storage setting is different because the client is the one that chooses the coding coefficients and enforces their use by the servers. Moreover, the solution in [55] leads to an increase in the size of the coded blocks after each encoding operation and cannot be used in a long-term storage setting where the number of repair operations is unbounded.

3.2 Background on Distributed Storage Systems

We give an overview of the main approaches proposed in distributed storage systems to store data redundantly across multiple storage servers: replication-based, erasure coding-based, and network coding-based. These are effective in a non-adversarial setting, where only benign faults may occur. For each, we outline the storage cost to store data redundantly and the network cost to restore the desired level of redundancy when a server fails. We also formulate the *data recovery condition*, which captures the amount of corruption that can be tolerated without affecting the ability to recover the original data. These approaches are illustrated in Figure 3.1.

We consider a file $\mathtt{F}$ that needs to be stored redundantly (we denote the size of $\mathtt{F}$ by $|\mathtt{F}|$). To express the network overhead of the repair component, we define the *network overhead factor* as the ratio between the amount of data that needs to be retrieved (from

healthy servers) to the amount of data that is created to be stored on a new server. This will be our primary metric to measure the communication cost of the repair component.

3.2.1 Replication

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

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

3.2.2 Erasure Coding

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

[1]Compared with replication, erasure coding achieves a reliability level that is an order of magnitude higher for the same redundancy level [56].

one new coded block, the client has to first reconstruct the entire file (*i.e.*, retrieve k coded blocks), thus incurring a network overhead factor of k.

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

3.2.3 Network Coding for Distributed Storage

Recent work in coding for distributed storage [18, 19] has shown that the k network overhead factor for the repair component is not unavoidable (as it was commonly believed). Given a file represented by m input blocks, $\bar{\mathtt{b}}_1, \bar{\mathtt{b}}_2, \ldots, \bar{\mathtt{b}}_m$, the client uses network coding to generate coded blocks as linear combinations of the original m file blocks. Each input block $\bar{\mathtt{b}}_i$ can be viewed as a column vector: $\bar{\mathtt{b}}_i = (b_{i1}, b_{i2}, \ldots, b_{iu})$, where b_{ij} are elements in a finite field $\mathbb{F}_{2^w}$ and are referred to as *symbols*.

Given a coding coefficient vector $(x_1, \ldots, x_m)$, in which x_i are chosen at random from $\mathbb{F}_{2^w}$, a coded block $\bar{\mathtt{c}}$ is computed as a linear combination of the input blocks:

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

The linear combinations of the symbols in the input blocks are performed over a finite field using randomly chosen coefficients. Thus, a coded block has the same size as an original file block and can also be viewed as a column vector $\bar{\mathtt{c}} = (c_1, \ldots, c_u)$. It has been shown [57, 58] that if the coding coefficients are chosen at random from a large enough field (*i.e.*, at least $\mathbb{F}_{2^8}$), then the original file can be recovered from m coded blocks by solving a system of m equations (because the m coded blocks will be linearly independent with high probability).

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

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

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

The storage cost is $n\alpha'$ bits across all servers (α' bits per server). The network overhead of the repair component is $\gamma' = \ell\beta'$ bits, so the network overhead factor is $\frac{\gamma'}{\alpha'}$. There is a tradeoff between the storage cost and the repair network overhead cost [19]. In short, for every tuple $(n, k, \ell, \alpha', \gamma')$, there exists a family of solutions which has two extremal points on the optimal tradeoff curve:

- One extremal point uses the pair $(\alpha', \gamma') = \left(\frac{|\mathtt{F}|}{k}, \frac{|\mathtt{F}|\ell}{k(\ell-k+1)}\right)$ to minimize the storage cost on the servers. It is referred to as a minimum storage regenerating (MSR) code. The

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

storage cost per server is $|\mathtt{F}|/k$, same as in the erasure coding-based approach[3], but this approach has a network overhead factor of $\frac{\ell}{\ell-k+1}$ and outperforms erasure coding in terms of network cost of the repair component whenever $\ell > k$.

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

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

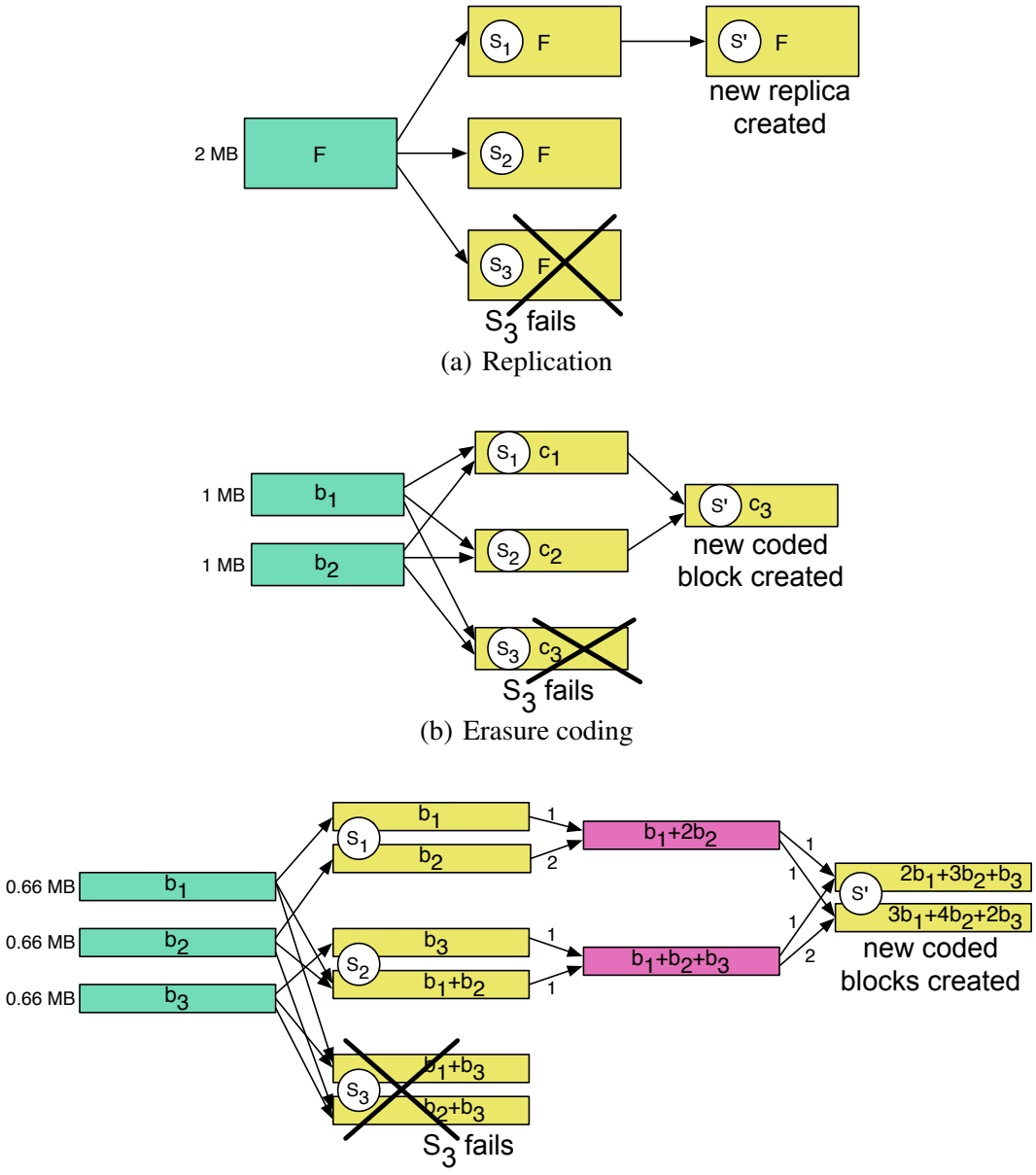
An Example. We illustrate the three approaches in Figure 3.1. Detailed explanations are as follows:

(a) In *replication*, copies of the file are stored at three servers, S_1, S_2, S_3. When the replica at S_3 gets corrupted, the client uses the non-corrupted replica at S_1 to create a new replica of size 2 MB. The client retrieves 2 MB in order to generate a new replica of size 2 MB, so the network overhead factor is 1.

(b) In *erasure coding*, the original file has two 1 MB blocks $(\mathtt{b}_1, \mathtt{b}_2)$ and is encoded into three blocks $(\mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3)$, using a $(3, 2)$ erasure code (so that $\mathtt{F}$ can be reconstructed from any two coded blocks). Each coded block is stored at a different server. When $\mathtt{c}_3$ gets corrupted, the client first retrieves $\mathtt{c}_1$ and $\mathtt{c}_2$ to reconstruct $\mathtt{F}$ and then regenerates the coded block $\mathtt{c}_3$. The client retrieves 2 MB in order to generate a new block of size 1 MB, so the network overhead factor is 2.

(c) In *network coding*, the original file has three 0.66 MB blocks and the client computes coded blocks as linear combinations of the original blocks. Two such coded blocks are stored on each of three storage servers. Note that this choice of parameters respects the guarantees of a $(3, 2)$ erasure code (*i.e.*, any two servers can be used to recover $\mathtt{F}$, because they will have at least three linearly independent equations, which allows to reconstruct the original blocks $\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3$). When the data at S_3 gets corrupted, the client uses the remaining two servers to create two new blocks: The client first retrieves one block from each healthy server (obtained as a linear combination of the server's blocks), and then further mixes these blocks (using linear combinations) to

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



(a) Replication

(b) Erasure coding

(c) Network coding

Figure 3.1 Example of various approaches for redundantly storing a file `F` of 2 MB.

obtain two new coded blocks which are stored at a new server. The numbers on the arrows represent the coefficients used for the linear combinations. The client retrieves 1.33 MB in order to generate a new coded block of size 1.33 MB, so the network overhead factor is 1.

3.3 System and Adversarial Model

Initially, the client stores data redundantly across a set of n storage servers, $S_1, S_2, \ldots, S_n$. We adopt an adversarial model similar to the one in HAIL [21]. We assume a *mobile adversary* that can behave arbitrarily (*i.e.*, exhibits Byzantine behavior) and can corrupt any (and potentially all) of the servers over the system lifetime. However, the adversary can corrupt at most $n - k$ out of the n servers within any given time interval. We refer to such a time interval as an *epoch*.

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

In every epoch, the adversary may choose a new set of $n - k$ servers and corrupt both the code and the storage component on these servers. At the end of each epoch, we assume that the code component of each server is restored to a correct state[4]. Although the code component is restored, the storage component may remain corrupted across epochs. Thus, in the absence of explicit defense mechanisms, the storage at more than $n-k$ servers may become corrupted and cause the original data to become unrecoverable. The client's goal is to detect and repair storage corruption before it renders the data unavailable. To this end, the client checks data possession with the servers in every epoch and if it detects faulty servers, it uses the redundancy at the remaining healthy servers to repair data at faulty servers.

An epoch consists of two phases:

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

1. A *challenge phase* that contains two sub-phases:
   (a) corruption sub-phase: The adversary corrupts up to b_1 servers.
   (b) challenge sub-phase: The client performs checks of data possession with the servers. As a result, the client may detect servers with corrupted storage (*i.e.*, *faulty* servers).

2. A *repair phase* that contains two sub-phases and is triggered only if corruption is detected during the challenge phase:
   (a) corruption sub-phase: The adversary corrupts up to b_2 servers.
   (b) repair sub-phase: The client repairs the data at any faulty servers detected in the challenge phase.

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

The structure of an epoch is similar with the one in [21], with one modification: We explicitly allow the adversary to corrupt data after the challenge phase. This models attackers that act honestly during the challenge phase, but are malicious in the repair phase.

3.4 RDC for Network Coding

In this section, we present our main RDC-NC scheme. To facilitate the exposition, we gradually introduce a series of challenges and our approaches to overcome them, leading to the main RDC-NC scheme in Section 3.4.4.

We consider remote data checking schemes for a storage system that relies on network coding to store and repair data as described in Section 3.2.3. The client chooses a set of parameters $(n, k, \ell, \alpha', \gamma')$ (in Section 3.6 we give guidelines on how to choose the parameters in a setting where the storage servers are untrusted). The file $\mathtt{F}$ is split into m blocks, $\bar{\mathtt{b}}_1, \ldots, \bar{\mathtt{b}}_m$. The client computes and stores $\alpha = \frac{\alpha' m}{|\mathbb{F}|}$ coded blocks at each of the n servers (*i.e.*, server S_i stores coded blocks $\bar{\mathtt{c}}_{i1}, \ldots, \bar{\mathtt{c}}_{i\alpha}$). A coded block is computed as a linear combination of the original m file blocks.

3.4.1 Can Existing RDC Schemes be Extended?

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

The challenge phase. Single-server RDC schemes compute verification metadata, which is stored together with the data and facilitates data integrity checks (Section 6.3.2). Such schemes can be extended to a multiple-server setting if we regard each coded block as a collection of *segments* and the client computes a challenge verification tag for each segment. A single-server RDC scheme based on spot checking (Section 6.3.2) can then be used to check integrity of each of the α blocks stored by each of the n servers.

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

In a distributed storage system that uses erasure coding to store a file redundantly across multiple servers, each of the n storage servers is assigned one erasure-coded block (*i.e.*, server S_i is assigned erasure-coded block i). For erasure coding, the layout of the encoded file is fixed and is known to the client (because the client knows the parity matrix

used for the erasure code). As a result, when coded block i is found corrupted, the *same exact* block i will be reconstructed by the repair phase. Thus, it is straightforward to embed the index of a block into challenge verification tags, as the client can use the same index i to challenge possession of the i-th block regardless of how many repair operations have occurred.

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

The repair phase. A malicious server may store the correct blocks it is supposed to store, and may act honestly in the challenge phase. However, the server may act maliciously in the repair phase, when it is asked to contribute coded blocks during the process of reconstructing blocks after a server failure. If the server does not combine its blocks correctly and contributes corrupted blocks, the corrupted blocks will lead to further corruption in the system as they are being combined with blocks from other servers. This *pollution* attack is possible because the client does not have access to the original blocks in order to check if the encoding operation was performed correctly by the server. Our RDC-NC scheme in Section 3.4.4 prevents pollution attacks in the repair phase by using a *repair verification tag* (different from challenge verification tags).

3.4.2 How to Maintain Constant Client Storage?

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

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

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

3.4.3 Replay Attacks

One concern with using logical identifiers is that a malicious server can reuse previously coded blocks with the same logical identifier (even old coded blocks that were previously

stored on failed servers) in order to successfully pass the integrity verification challenge. The data recovery condition could be broken if the malicious server is able to find and reuse old coded blocks such that they are linearly dependent with other coded blocks that are currently stored across servers. In essence, this is a *replay attack* that could potentially lead to breaking the data recovery condition. In Appendix B.1, we give a concrete replay attack example for a configuration similar with the one in Figure 3.1.

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

A more efficient solution to mitigate replay attacks. While the example in Appendix B.1 shows that successful replay attacks are possible, the example is somewhat artificial in that the coding coefficients used in the repair phase are specifically chosen in order to break the data recovery condition. In fact, replaying old coded blocks is only harmful if it introduces additional linear dependencies among the coded blocks stored at the servers (*i.e.*, entropy

is reduced). Otherwise, replay attacks are not harmful, because the underlying principle that ensures the validity of network coding is preserved.

We mitigate replay attacks by adopting a very simple but effective solution: In traditional network coding, the coding coefficients are stored in plaintext at the servers together with their corresponding coded blocks (note that to maintain constant client storage, the coding coefficients should always be stored in the servers, rather than in the client). Instead, we store the coding coefficients encrypted in the servers, which will prevent the adversary from knowing how the original blocks were combined to obtain the coded block. Thus, even if the adversary corrupts servers, its ability to execute harmful replay attacks is negligible, as it does not have the necessary knowledge to target certain blocks on certain servers or to know which old coded blocks to replay. Theorem 3.4.1 shows that by encrypting the coding coefficients, a malicious server's ability to execute a harmful replay attack becomes negligible.

Theorem 3.4.1. *We consider a network coding-based distributed storage system in which any k out of n servers can recover a file with high probability. Let P_1 be the probability to recover the file in a setting where the coding coefficients are stored in plaintext and the storage servers are not malicious (*i.e. *a benign setting). Let P_2 be the probability to recover the file in a setting where the coding coefficients are stored encrypted and the adversary can corrupt storage servers and execute replay attacks. Then, $|P_1 - P_2| \leq \epsilon$, where $\epsilon \to 0$ (*i.e.*, these probabilities are negligibly close).*

Proof. (sketch) In Fig. 3.2, we use a representation of the distributed storage system similar with the information flow graph used in [18, 19] to describe how encoded data is stored at storage nodes and how data is transmitted for the purpose of repairing faulty storage nodes

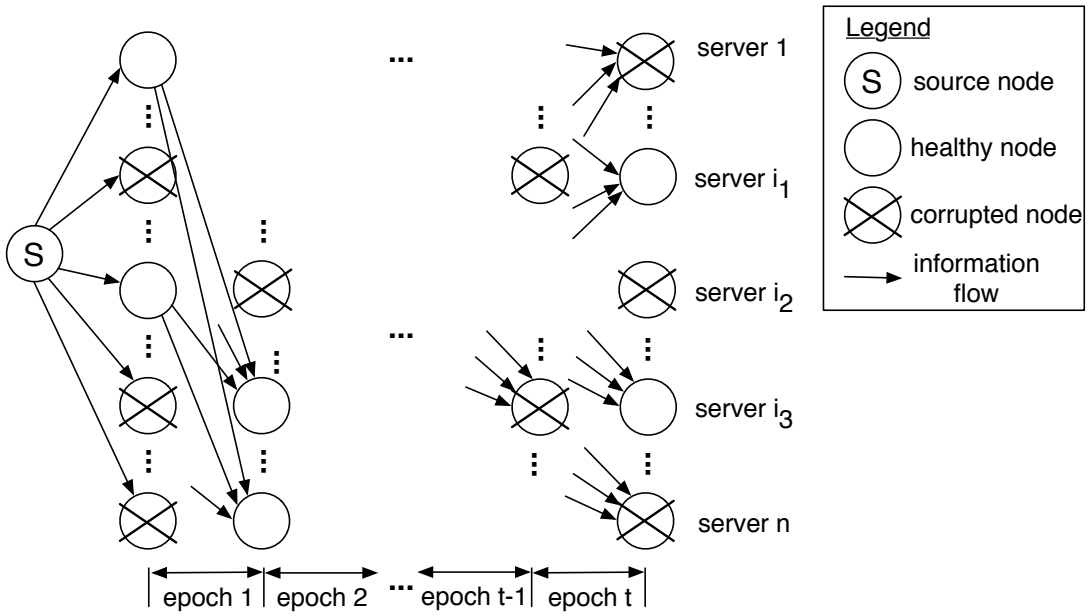


Figure 3.2 An illustration of the information flow graph after t epochs. A node in this graph represents the storage at a specific server in a particular epoch. The source node S has the original file, which is then encoded using network coding and stored at n servers. In each epoch, the data on at most $n - k$ servers can be corrupted (due to either benign or adversarial faults). At the end of each epoch, the servers with corrupted data are detected and repaired using data from k healthy servers. An information flow arrow incoming into a node in epoch i means that the node is repaired at the end of epoch i using data from healthy nodes.

and for recovering the original data. In each epoch the data on at most $n - k$ servers can be corrupted due to benign faults or due to the attacker. At the end of each epoch, the servers with corrupted data are detected and repaired using data from k healthy servers.

We first consider the benign setting, where only non-adversarial faults may occur. The system guarantees that the original file can be recovered from any k out of the n nodes with high probability. This guarantee holds for any epoch, after the repair phase. Equivalently, there are $\binom{n}{k}$ receivers [58] that can recover the file (*i.e.*, receiver 1 can recover the file from servers $1, \ldots, k$, receiver 2 can recover the file from servers $1, \ldots, k-1, k+1$, etc).

We then consider a setting in which data may be corrupted due to both non-adversarial or adversarial faults (*i.e.*, servers may fail due to benign faults or the adversary

may target to corrupt data on specific servers). Corrupted data is detected and repaired in each epoch. However, the attacker records all data prior to corruption and accumulates all old coded blocks. Let N_t denote the number of all nodes in the information flow graph after t epochs, except for the source node (this includes both healthy and corrupted nodes). Thus, in epoch $t+1$, the attacker can access data from N_t nodes to execute the replay attack. We distinguish between two cases:

– *Unencrypted coefficients*. The coding coefficients are stored in plaintext together with the coded data. We now show that, at the end of epoch t, the system must guarantee that any k nodes out of N_t can recover the file, under the condition that the k nodes belong to different servers (since the logical identifier ensures that a node can only be used for a specific server). If, at the end of epoch t, there exists at least one set of k nodes out of N_t which belong to different servers and which do not have enough information to recover the file, then the attacker can use this set to cause permanent damage (we refer to this one set as the "bad" set). Then, over the course of at most $\lceil \frac{k}{n-k} \rceil$ epochs ($n-k$ servers per epoch), the attacker can execute replay attacks and gradually replace current data on the k servers corresponding to the nodes in the "bad" set with old data from the nodes in the "bad" set. The replay attacks will not be detected. Then, in the next epoch, the attacker corrupts data on the other $n-k$ servers, causing permanent file damage, since the k servers corresponding to the "bad" set do not have enough information to recover the file. Equivalently, there are $\binom{N_t}{k} - f(t)$ receivers that should be able to recover the file, where $f(t)$ represents the number of receiver that are connected to nodes that belong to the same server. We have $(t-1)(\binom{n}{k}-1)+\binom{n}{k} \leq \binom{N_t}{k} - f(t) \leq t^n\binom{n}{k}$. According to Theorem 1 in [58], for a fixed field size, a system based on random linear network coding has an upper bound on the number of receivers it can support. Thus, since $\binom{N_t}{k} - f(t)$ grows unbounded as t grows, the system cannot guarantee that the original file can be recovered.
– *Encrypted coefficients.* The coding coefficients are stored encrypted together with the coded data. Since the attacker has no knowledge of the coding coefficients, it has no better strategy than picking at random data it has stored from nodes that were corrupted in the past, and using this data to replace data currently stored on servers. Note that replay attacks remain undetectable only if old data from server i is replayed on the same server i, because of the logical identifiers embedded in the challenge verification tags. This means that, unlike in the case of unencrypted coefficients, the system should only guarantee that at the end of epoch t the file can be recovered from any k out of n nodes. But this is the same guarantee that is already achieved by the system in the benign case! Intuitively, the guarantee is preserved because, even if there exists a "bad" set of k nodes (belonging to k different servers), the attacker cannot identify these nodes (as the coefficients are encrypted) and can only pick nodes at random for the replay attack.

□

In Appendix B.2, we provide a simulation to further validate Theorem 3.4.1. Thus, we conclude that when coding coefficients are encrypted, the replay attack is a negligible threat. As a result, the client does not need to take other explicit countermeasures to mitigate the replay attack, besides encrypting the coefficients.

Additionally, since the malicious servers will help generate the data in the repair phase, and they can simply use the coefficients which are not random enough to generate the data, so that the entropy of the network coding-based storage systems may be reduced (*i.e.*, entropy attack). To mitigate such an attack, in the repair phase, the client chooses the random coefficients that should be used by the servers and enforces their use by the servers. For the remainder of this section, we will give a solution in which the coding coefficients are encrypted before being stored on the servers, and the client picks the random coefficients, and enforces the servers to use them during repair.

3.4.4 Remote Data Checking for Network Coding (RDC-NC)

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

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

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

- For the purpose of checking data possession (in the challenge phase), a block (either original or coded) is viewed as an ordered collection of s segments. For example a

coded block $\bar{\mathtt{c}}_{ij} = (\mathtt{c}_{ij1}, \ldots, \mathtt{c}_{ijs})$, where each segment $\mathtt{c}_{ijk}$ is a contiguous portion of the block $\bar{\mathtt{c}}_{ij}$ (in fact, each segment contains one symbol).

- For the purpose of network coding, a block (either original or coded) is viewed as a column vector of u symbols (as described in Section 3.2.3). For example, a coded block $\bar{\mathtt{c}}_{ij} = (c_{ij1}, \ldots, c_{iju})$, where $c_{ijk} \in \mathbb{F}_p$[5].

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

To detect direct data corruption attacks, the client performs a spot checking-based challenge similar as in POR [10] and PDP [7] to check the integrity of each network coded block stored by each of the n servers. The challenge tag for a segment in a coded block binds the data in the segment with the block's logical identifier and also with the coefficient vector that was used to obtain that block. Thus, the client implicitly verifies that the server cannot use segments from a block with a different logical identifier to pass the challenge, and also that the coefficient vector retrieved by the client corresponds to the block used by the server to pass the challenge. If a faulty server is found in the challenge phase, the client uses the remaining healthy servers to construct new coded blocks in the repair phase and stores them on a new server.

The details of RDC-NC scheme are presented in Figures 3.3, 3.4, and 3.5. We rely on a construction of a message authentication code which, as indicated by Bowers et al. [21],

[5]Note that, unlike the description for network coding in Section 3.2.3 which uses symbols from $\mathbb{F}_{2^w}$, we use symbols from the finite field $\mathbb{F}_p$ over the integers modulo p, where p is a prime such that $\mathbb{F}_p$ and $\mathbb{F}_{2^w}$ have about the same order (*e.g.*, when $w = 8$, p is 257). Based on a similar analysis as in [55], the successful decoding probability for network coding under $\mathbb{F}_{2^w}$ and under $\mathbb{F}_p$ is similar.

has been proposed as a combination of universal hashing with a PRF [59–62]. Shacham and Waters [10] use a similar construction. We adapt this construction to our network coding-based distributed storage setting.

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

In GenBlockAndMetadata, the client picks random coefficients from $\mathbb{F}_p$ and uses them to compute a new coded block $\bar{c}_{ij}$ (steps 2 and 3). For each segment in the coded block, the client computes a challenge tag which is stored at the server and will be used by the server in the Challenge phase to prove data possession. For each segment in the coded block $\bar{c}_{ij}$, the client embeds into the challenge tags of that segment the coefficient vector used to obtain $\bar{c}_{ij}$ from the original file blocks (step 4). For example, in Figure 3.1(c), the second block stored at server S_1 has been computed using the coefficient vector $[0, 1, 0]$ (and its logical identifier is "1.2"). Thus, the challenge tag for the k-th segment in this block is[6]:

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

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

Recall from Section 3.2.3 that $\alpha = \alpha' m/|\mathtt{F}|$ and $\beta = \beta' m/|\mathtt{F}|$. We construct a network coding-based RDC scheme in three phases, Setup, Challenge, and Repair. Let $f : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow \mathbb{F}_p$ be a PRF and let $(\mathsf{G}, \mathsf{E}, \mathsf{D})$ be a semantically secure encryption scheme. We work over the field $\mathbb{F}_p$ of integers modulo p, where p is a large prime (at least 80 bits).

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

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

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

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

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

Challenge: For each of the n servers, the client checks possession of each of the α coded blocks stored at each server (by using spot checking of segments for each coded block). In this process, each server uses its stored blocks and the corresponding challenge tags to prove data possession.
For $1 \leq i \leq n$:

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

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

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

Figure 3.3 RDC-NC: Setup and Challenge phase.

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

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

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

(a) C generates a set of coefficients $(x_1, \ldots, x_\alpha)$, where $x_k \xleftarrow{R} \mathbb{F}_p$, with $1 \leq k \leq \alpha$

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

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

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

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

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

7. Generate a value δ using f and K_{prf1}: $\delta = f_{K_{prf1}}(y)$ (the δ value will be used for generating the challenge tags)
Generate u values λ_1, ..., λ_u using f and K_{prf2}: $\lambda_k = f_{K_{prf2}}(y||k)$, with $1 \leq k \leq u$ (the λ values will be used for generating the repair tag)
For $1 \leq j \leq \alpha$:
   - C runs $(\bar{\mathtt{c}}_{yj}, z_{yj1}, \ldots, z_{yjm}, \mathtt{t}_{yj1}, \ldots, \mathtt{t}_{yjs}, \mathtt{T}_{yj}) \leftarrow \mathsf{GenBlockAndMetadata}(\bar{\mathtt{a}}_{i_1}, \ldots, \bar{\mathtt{a}}_{i_\ell},$ "$y.j$", $\delta, \lambda_1, \ldots, \lambda_u, \mathrm{key})$
   - For $1 \leq k \leq m$: $\epsilon_{yjk} = \mathsf{E}_{K_{enc}}(z_{yjk})$ //encrypt coefficients $z_{yj1}, \ldots, z_{yjm}$

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

Figure 3.4 RDC-NC: Repair phase.

The Challenge **phase.** For this phase, we rely on the scheme in [10], adapted as described in Sections 3.4.1 and 3.4.2 (in short, the challenge tags include the logical identifier of the block, the index of the segment, and the coding coefficients used to obtain that block). Note that we rely on the scheme in [10] that allows private verifiability (*i.e.*, only the data owner can check possession of the data). However, RDC schemes that allow public verifiability [7, 10] could also be adapted following a similar strategy.

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

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

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

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

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

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

Figure 3.5 Components of the RDC-NC scheme.

The Repair **phase.** The client contacts ℓ healthy servers $S_{i_1}, \ldots, S_{i_\ell}$ and asks each one of them to generate a new coded block (step 1)[7]. The client further combines these ℓ coded blocks to generate α new coded blocks and metadata (step 2), and then stores them on a new server S' (step 3). As part of step 1, for each contacted server, the client chooses random coefficients from $\mathbb{F}_p$ that should be used by the server to generate the new coded

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

block (step 1(a)). Each contacted server, based on its α stored blocks and on the coefficients provided by the client, runs GenRepairBlock (step 1(c)) to compute a new coded block $\bar{\mathsf{a}}_i$ (GenRepairBlock, step 1) and a proof of correct encoding τ_i (GenRepairBlock, step 2); the server then sends these to the client, together with the encrypted coefficients corresponding to the blocks used to compute the new coded block. The client decrypts the coefficients (step 1(d)), re-generates the λ values used to compute the repair tags (step 1(e)), and then checks whether the encoding was done correctly by the server (step 1(f)). As a result, the client is ensured that the server has computed the new coded block by using the correct blocks and the coefficients supplied by the client.

3.5 Analyses for RDC-NC

A tradeoff between storage and communication. In the RDC-NC scheme described in Figures 3.3, 3.4, and 3.5, each segment contains only one symbol (*i.e.*, an element from $\mathbb{F}_p$) and is accompanied by a challenge tag of equal length. The scheme can be modified to use a similar strategy as in [10]: Each segment can contain r symbols, which reduces the server storage overhead by a factor of r and increases the client-server communication overhead of the challenge phase by a factor of r.

Protection against small data corruption. A spot checking mechanism for the challenge phase is only effective in detecting "large" data corruption [7,9]. To protect against "small" data corruption, we can combine the RDC-NC scheme with error correction codes such as a "server code" [17, 21] or a method to add "robustness" [16]. The client applies a server code on the network coded blocks before computing the challenge and repair tags. In the repair phase, a server does not include the server code portion when computing new coded

blocks. Instead, the client computes the server code for the new coded blocks. We leave as future work the design of schemes in which the server code portion can be computed together with the rest of the block using network coding.

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

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

Theorem 3.5.1. *The data recovery condition is a sufficient condition to ensure data recoverability in the* RDC-NC *scheme augmented with protection against small data corruption.*

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

In any given epoch, the adversary can corrupt at most $n - k$ servers. The adversary may split the corruptions between direct data corruptions and replay attacks. Faulty servers affected by direct data corruptions are detected by the integrity checks in the challenge phase, or by the correct encoding check in the repair phase. The client uses the remaining healthy servers (at least k remain healthy) to regenerate new coded blocks to be stored on new servers. Protection against small data corruption is provided by the server code layer. From Theorem 3.4.1, replay attacks against RDC-NC are not harmful, and they do

not increase the attacker's advantage in breaking the data recovery condition. The check in Repair, step 1(f), ensures protection against pollution attacks. Thus, at the end of the epoch, the system is restored to a state equivalent to its initial state, in which the file can be recovered from any k out of the n servers. □

3.6 Guidelines for Choosing Parameters for RDC-NC

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

Based on the desired reliability level (which is a function of perceived fault rate of the storage environment), the client picks the values for n and k. We focus on MBR codes, which are characterized by the pair $(\alpha', \gamma') = \left(\frac{2|\mathbb{F}|\ell}{2k\ell-k^2+k}, \frac{2|\mathbb{F}|\ell}{2k\ell-k^2+k}\right)$ and which minimize the network overhead factor of the repair phase (*i.e.*, $\gamma'/\alpha' = 1$). Although we give guidelines on choosing parameters for MBR codes, our RDC-NC scheme is general and can be applied to any parameter tuple.

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

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

$$\alpha = \frac{2\ell m}{2k\ell - k^2 + k} \tag{3.1}$$

Even though we have fixed a point characterized by a minimal network overhead factor of 1, different values of the parameters α, m, and ℓ will result in different storage overheads at the servers. We now show that by choosing $\ell = k$, we minimize the total storage across the n servers (recall that ℓ is the number of healthy servers that are contacted by the client in the repair phase). We need to provision the system for the worst case, in which the adversary corrupts $n - k$ servers in some epoch. In this case, the maximum number of servers that remain healthy in that epoch is k. Thus, we need to set $\ell \leq k$. Minimizing the total storage across the n servers means we need to minimize the quantity $n\alpha' = n|\mathtt{F}|\frac{\alpha}{m}$. Since n is fixed, for a given file size $|\mathtt{F}|$, we should minimize $\frac{\alpha}{m}$. From Eq. (3.1), $\frac{\alpha}{m} = \frac{2\ell}{2k\ell - k^2 + k} = \frac{2}{2k - \frac{k^2 - k}{\ell}}$, which is minimized when ℓ is maximized. Thus, we need to set $\ell = k$.

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

$$m = \frac{\beta(2k\ell - k^2 + k)}{2} = \frac{\beta(k^2 + k)}{2} \tag{3.2}$$

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

$$\alpha = \ell\beta = k\beta \tag{3.3}$$

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

3.7 Experimental Evaluation

In this section, we evaluate the computational performance of our RDC-NC scheme (the analysis of its communication and storage overhead can be found in Table 3.1 of Section 3.0.1).

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

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

Table 3.2 Experimental Test Cases

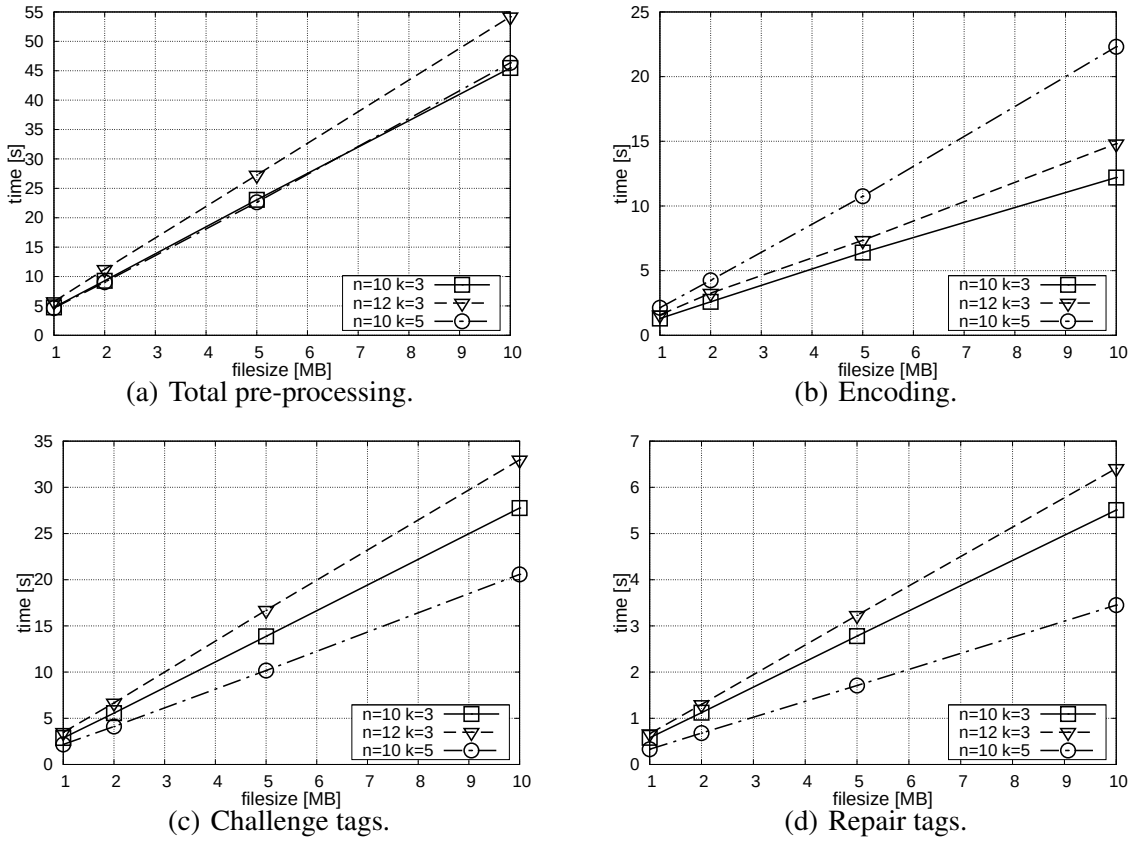
$case$	n	k	α	m	l	β
1	10	3	3	6	3	1
2	12	3	3	6	3	1
3	10	5	5	15	5	1

When evaluating the performance of RDC-NC, we are only concerned with pre-processing (in the setup phase) and the repair phase (the performance of the challenge phase was evaluated in [7]).

We choose to evaluate three cases, as shown in Table 3.2. They satisfy the condition of MBR codes with minimal server storage (see Section 3.6). We use $\beta = 1$.

3.7.1 Pre-Processing Phase Results

The client pre-processes the file before outsourcing it. Specifically, the client: (a) encodes the m-block file using network coding over $\mathbb{F}_p$, generating $n\alpha$ coded blocks (α blocks for each server), (b) computes the challenge tags, and (c) computes the repair tags. Figure 3.6 shows the computational cost of client pre-processing and its various components for different file sizes under the three test cases. Note that the amount of data that needs to be pre-processed can be significantly larger than the original file size. For example, for test case 1 ($n = 10, \alpha = 3, m = 6$), if $|\mathtt{F}| = 10$MB then the client has to pre-process $|\mathtt{F}|\alpha n/m$ = 50MB. We can see that these computational costs are all increasing linearly with the file size. The cost of generating challenge and repair tags is determined by the total amount of pre-processed data $n\alpha'$, which can be further expressed as $\frac{2n*|\mathtt{F}|}{k+1}$ ($\alpha' = \frac{2*|\mathtt{F}|}{k+1}$, inferred from Section 3.6). Thus, for fixed $|\mathtt{F}|$, the cost for generating challenge and repair tags increases with n and decreases with k.



(a) Total pre-processing. (b) Encoding.

(c) Challenge tags. (d) Repair tags.

Figure 3.6 Computational cost for client pre-processing and its various components (to pre-process data for n servers).

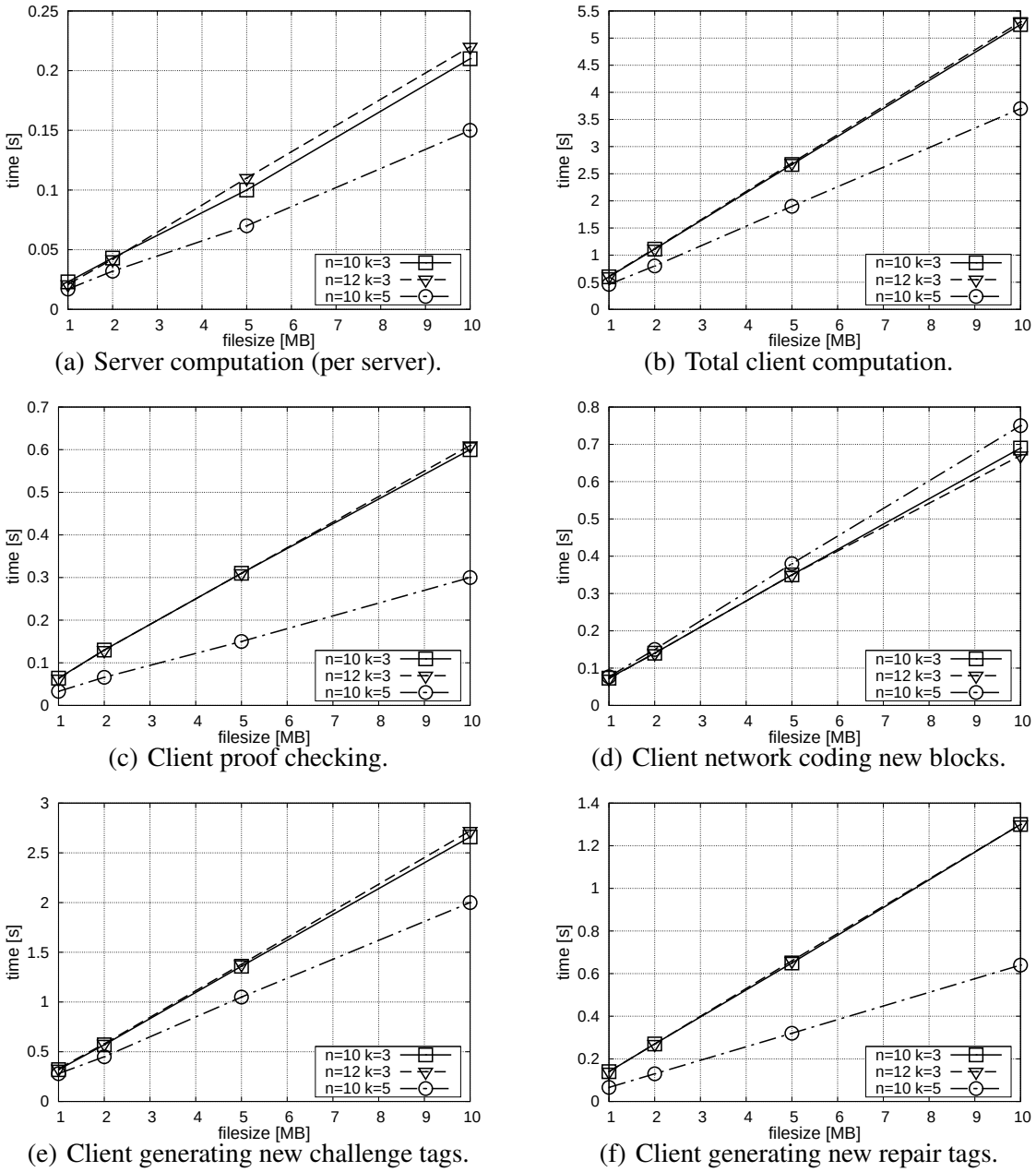
3.7.2 Repair Phase Results

We assume that the client needs to repair one server.

Server computation. In the repair phase, the client retrieves blocks from ℓ servers. Every server generates β new blocks, together with the aggregation of the tags using the coefficients sent by client. Figure 3.7(a) shows the per-server computational cost under the three test cases. The computational cost increases linearly with the file size for all test cases. We observe that the per server computation varies by k, rather than by n; specifically, for fixed file size, when k increases, per server computation decreases. This can be explained as follows. Per server computation contains two components, encoding and aggregating the tags, and is dominated by the encoding component. The cost of encoding is $O(\frac{\beta\alpha|\mathbb{F}|}{m})$,

which can be further expressed as $O(\frac{2*|\mathbb{F}|}{k+1})$ (cf.Section 3.6, with $\beta = 1$). Thus, for fixed $|\mathbb{F}|$, per server computation is only determined by k in a monotonically decreasing way.

Client computation. After getting β blocks from each of the ℓ servers, the client checks the proof sent by each server. The client then generates α new coded blocks from these $\beta\ell$ blocks (using random network coding over $\mathbb{F}_p$), together with the challenge tags and repair tags. Figures 3.7(b)-3.7(f) show the total client computational cost to repair one server and the costs of its various components. These figures show that the client computational cost is linear to the file size and varies by k, rather than by n. The storage at one server, $\frac{2*|\mathbb{F}|}{k+1}$, determines the cost of proof checking, generating challenge tags and repair tags, thus can explain why computational costs of these components are decreasing with k for fixed file size. The cost of encoding in the repair phase is $O(\frac{\alpha\beta\ell|\mathbb{F}|}{m})$, *i.e.*, $O(\frac{|\mathbb{F}|}{1+1/k})$ (cf. Section 3.6, with $\beta = 1$), thus, encoding cost increases with k for fixed file size.



(a) Server computation (per server).

(b) Total client computation.

(c) Client proof checking.

(d) Client network coding new blocks.

(e) Client generating new challenge tags.

(f) Client generating new repair tags.

Figure 3.7 The computational cost of the repair phase: (a) server cost, (b)-(f) client cost to repair one server.

CHAPTER 4

TOWARDS SELF-REPAIRING REPLICATION-BASED STORAGE SYSTEMS USING UNTRUSTED CLOUDS

In Chapter 3, we have introduced RDC-NC, a remote data checking scheme for network coding-based distributed storage systems, in which we investigate RDC in a multiple-server setting (specifically, the original file is first encoded by network coding, then the encoded data is distributed to multiple untrusted servers). The setting considered so far outsources the storage of the data, but the data owner is still heavily involved in the data management process (especially during the repair of damaged data). In this chapter, we propose a new paradigm, in which the data owner fully outsources both the data storage and the management of the data. Specifically, we enable the self-repairing functionality in the cloud servers, so that the client can get liberated from the burden of repair and thus is able to be kept lightweight.

The recent proliferation of cloud services has made it easier than ever to build distributed storage systems based on Cloud Storage Providers (CSPs). Traditionally, a distributed storage system stores data redundantly at multiple servers which are geographically spread throughout the world. In a benign setting where the storage servers always behave in a non-adversarial manner, this basic approach would be sufficient in order to deal with server failure due to natural faults. In this work, we consider a setting in which the storage servers are untrusted and may act maliciously. In this setting, Remote Data Checking (RDC) [7–10] 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 [21,22], 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

([21, 22]). 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 chapter, we ask the question: Is it possible to design an RDC scheme which can repair corrupted data with the least data owner intervention? We answer this question in the positive by exploring a model which minimizes the data owner's involvement in the Repair phase, thus fully realizing the vision of outsourcing both the storage and management of data. During Repair, the data owner simply acts as a *repair coordinator*, which allows the data owner to manage data using a lightweight device. This is in contrast with previous work, which imposes a heavy burden on the data owner during Repair. The main challenge is how to ensure that the untrusted servers manage the data properly over time (*i.e.*, take necessary actions to maintain the desired level of redundancy when some of the replicas have failed).

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

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

The first two properties alone can be achieved based on techniques proposed in previous work ([22] provides multiple replica guarantees, whereas RDC based on spot-checking [8–10] supports a lightweight verification mechanism in the Challenge phase). The challenge is to achieve the third property without giving up any of the first two properties. We meet these objectives by proposing a new model and by redesigning the three phases of a traditional RDC protocol.

4.0.3 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 [63, 64] 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 4.2.1 we show that this approach has major limitations. Moreover, we show that for real-world CSPs, one of the assumptions made by [63] 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 C.1 (Appendix C.1) 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 ([22]) 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 and the means to generate new replicas. This will allow the servers to generate a new replica by collaborating between themselves during Repair.

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

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

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

4.1 Related Work

4.1.1 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) [8] and Proofs of Retrievability (PoR) [9, 10]. Later RDC schemes investigated models that can provide strong integrity guarantees while supporting dynamic operations on the outsourced data [11–15, 24, 25, 65–67]. Recently, several RDC schemes have been designed to secure outsourced version control systems [27, 68].

RDCs for the multiple-server setting. RDC has been extended to the multiple-server setting (distributed RDC). Curtmola et al. proposed MR-PDP [22], 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. [21] and Wang et al. [28] built RDC schemes for erasure coding-based distributed storage systems. Chen et al. [23] proposed an RDC scheme for network coding-based distributed storage systems.

All the aforementioned distributed RDCs adopt client-side repair, in which the client is intensively involved in the repair procedure, *i.e.*, the client will retrieve the data, generate and upload the new data to repair the corruption. Our work proposes server-side repair, a novel strategy which is different from all the previous distributed RDCs.

A new direction for RDC. All the previous RDC schemes are cryptography-based, *i.e.*, the security of the proposed schemes are inherited from the security of the cryptographic primitives. Bowers et al. [69] propose RAFT, a new time-based RDC scheme which can enable a client to obtain a proof that a given file is distributed across an expected number of physical storage devices in a single datacenter.

Although RAFT and our work share the idea of using a time-based mechanism to detect malicious behavior, they are fundamentally different in their basic approach and goals, and in the system and adversarial models. *First*, while in 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. [63] propose another time-based model (BDS model) to guarantee that multiple replicas are distributed to different data centers of the CSP. Our work adapts this model to enable the server-side repair.

Watson et al. [70] propose LoSt, which formalizes the concept of Proofs of Location (PoL). A PoL relies on a geolocation scheme [63] 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 basic 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 [71] further relax the adversarial models and assumptions of previous PoL scheme [63], and propose a constraint-based data geolocation protocol that binds the latency-based geolocation techniques with PDP scheme.

4.1.2 Proofs of Work (PoW)

CPU-bound PoW. Dwork and Naor [72] pioneer the concept of proofs of work, which is originally proposed to discourage junk emails. The basic idea is two-fold: firstly, 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; secondly, the receiver can efficiently verify $f(m)$, *i.e.*, verifying $f(m)$ is a lot faster than computing $f(m)$. They suggest some CPU-intensive candidates for function $f()$, *e.g.*, a function based on the signature scheme of Fiat and Shamir [73]. RDC-SR is in effect a CPU-bound PoW, in which we assume the CPU capability is bounded.

Memory-bound PoW and proofs of space. Abadi et al. [74] observe 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 investigate an alternative of PoW which measures the number of times the memory is accessed. Their approach is further improved in [75–77]. Recently, Dziembowski et al. [78] and Ateniese at al. [79] independently propose 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 is pointed out in [79], the schemes proposed by Dziembowski et al. [78] are in effect a variant of memory-bound PoWs, since their prover can possibly trade off space with computation. RDC-SR scheme relies 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 [74] 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 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. [80], 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, their work is set in the context of P2P networks and the verifier (client) needs to keep the data for the verification purpose.

4.2 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.

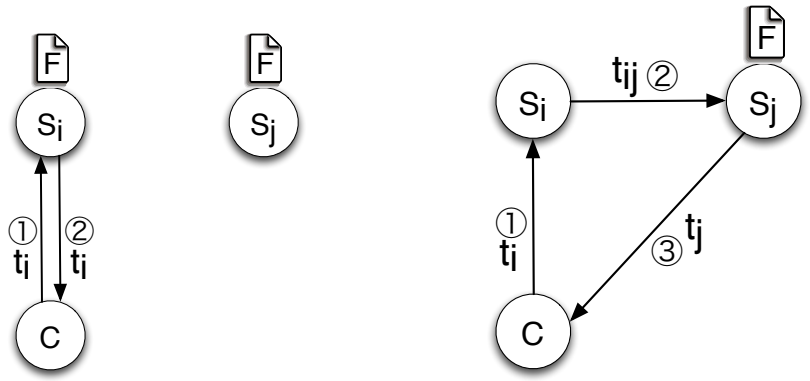
4.2.1 A Network Delay-based Model and Its Limitations

Benson, Dowsley and Shacham proposed a theoretical model for verifying that copies of a client's data are stored at different geographic locations [63] (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 [63]. The customer (client) makes a contract with the CSP to store one copy of the client's file in each of the CSP's k data centers. For simplicity, if we assume that $k = 2$, then a file copy should be stored at s_i and another file copy at s_j. The goal is to build an audit protocol that tests if the cloud provider is really storing one copy of the file in each of the two data centers s_i and s_j. Several assumptions need to be made:

(Assumption 1) The locations of all data centers of the cloud provider are known.
(Assumption 2) The cloud provider does not have any exclusive Internet connection between the data centers.
(Assumption 3) For each datacenter s, it is possible to have access to a machine that is located very close to s (*i.e.*, very small network latency), such as in the same data center.

Consider the case when the client wants to check if s_i is storing a copy of the file. As shown in Figure 4.1(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.



(a) Benign server behavior.

(b) Malicious server behavior.

Figure 4.1 Auditing protocol: Client C checks if server s_i has a file copy F.

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

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

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

$$T_i \leq min(t_i) + min(t_{ij}) + min(t_j) - 2 * max(t_i) \tag{4.1}$$

Limitations of the basic PoR protocol based on BDS model. Based on the condition derived from the BDS model, [63] 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 C.1 and C.2 in Appendix C.1). 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 C.1, 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 C.2 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 C.1 and C.2, with s_i and s_j being the Virginia and the N. California data centers respectively, and assuming that the auditor is located within s_i and challenges k 4KB random file blocks from s_i, Equation (4.1) for the basic PoR protocol becomes $x \cdot k \leq 80 + 0.3k$, where x is the time to access one random file block. According to our experiments on Amazon S3, $x \approx 30$ milliseconds (refer to Appendix C.2), thus $k \leq 2.66$. This means the basic PoR protocol applied in the setting of the Amazon CSP allows the auditor to challenge at most two random file blocks in each protocol execution. This provides a very low data possession assurance (comparatively, to achieve 99% confidence that misbehavior will be detected when the server corrupts 1% of the file, the auditor should challenge 460 randomly chosen file blocks [8]).

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 4.2.1) 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 4.2.1 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 4.2.1, an audit protocol execution should be considered invalid if the answer is received after $min(t_i) + min(t_{ij}) + min(t_j) + min(t_R)$, where t_R denotes the time required to generate replica $\mathtt{F}_i$ (more precisely, the time required to generate the portion of $\mathtt{F}_i$ that is necessary to construct

a correct answer). Thus, the condition used to differentiate benign from malicious behavior becomes:

$$T_i + 2 * max(t_i) \leq min(t_i) + min(t_{ij}) + min(t_j) + min(t_R) \tag{4.2}$$

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

(Assumption 1) The locations of all datacenters of the cloud provider are known.
(Assumption 2) For each datacenter s, it is possible to have access to a machine that is located very close to s (*i.e.*, very small network latency), such as in the same data center.

4.3 System and Adversarial Model

4.3.1 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 $\mathtt{F}_j$.

4.3.2 Adversarial Model

We assume that the CSP is rational and economically motivated. The CSP will try to cheat only if cheating cannot be detected and if it achieves some economic benefit, such as using less storage than required by contract. An economically motivated adversary captures many practical settings in which malicious servers will not cheat and risk their reputation, unless they can achieve a clear financial gain. We also note that when the adversary is fully malicious, *i.e.*, it tries to corrupt the client's data without regard to its own resource consumption, there is no solution to the problem of building a reliable system with t replicas [22, 69].

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

To illustrate the ROTF attack, consider the setting in Figure 4.1(b), where s_i and s_j should to store replicas $\mathtt{F}_i$ and $\mathtt{F}_j$, respectively, but only s_j stores $\mathtt{F}_j$. When s_i is being challenged to prove possession of $\mathtt{F}_i$, s_i can retrieve $\mathtt{F}_j$ from s_j, and generate $\mathtt{F}_i$ on the fly in order to pass the challenge. Or, it can forward the challenge to s_j, who uses $\mathtt{F}_j$

to generate on the fly $\mathtt{F}_i$ and then uses $\mathtt{F}_i$ to construct a valid response to the challenge. Immediately after the challenge, $\mathtt{F}_i$ is deleted.

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

Definition 4.3.1. *An* α-cheating adversary *is an economically-motivated adversary that can successfully pass a challenge by only using* $\alpha t|F|$ *storage (where* $\frac{1}{t} \leq \alpha \leq 1$*).*

Note that if $\alpha < \frac{1}{t}$, then the CSP stores less than $|\mathtt{F}|$, which means that any single-replica RDC scheme [8, 9] would be enough to detect the CSP's dishonest behavior. Thus, we do not consider the case when $\alpha < \frac{1}{t}$.

We consider a **static α-cheating adversary**, which refers to an α-cheating CSP, who possesses a fixed amount of computational power known by the client, and cannot grow its computational power over time. This captures a setting that the CSP initially has a fixed budget for the computational power, and will not increase its budget over time.

Adversarial Strategies Replica generation in our model is time consuming. A dishonest CSP trying to cheat by storing less replicas and executing the ROTF attack is always better off by keeping a copy of the original file $\mathtt{F}$. 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 [8–10]: 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 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.

It turns out that the *best data distribution strategy for an α-cheating adversary* is when the adversary stores in each of the t servers an equal fraction of the whole storage (Theorem 4.3.2), *i.e.*, $\alpha|F|$ storage at each server. Thus, the adversary will still only use $\alpha t|\mathtt{F}|$ storage space in total[1]. When any of the t servers is challenged, this server will already possess, on average, an α fraction of the c blocks that are being challenged (assume

[1]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 $\alpha t|\mathtt{F}|$.

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 4.3.2. *For an α-cheating adversary, the best data distribution strategy is to store in each of the t servers an equal fraction of the whole $\alpha t|F|$ storage.*

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 $1 \leq i \leq t$. 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, $P(A) = P(A_1 \cap A_2 \cap \ldots \cap A_t)$. In our setting, A_1, A_2, $\ldots$, A_t can be seen as independent events, thus:

$$P(A) = P(A_1)P(A_2)\ldots P(A_t) \tag{4.3}$$

Let x_i denote the number of blocks actually stored in server i, *i.e.*, $n - x_i$ blocks are missing in that server. We have:

$$\sum_{i=1}^{t}(x_i) = \alpha n t \tag{4.4}$$

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 PDP [8], we

have:

$$P(A_i) = \frac{x_i + \lambda\tau}{n} \frac{x_i + \lambda\tau - 1}{n-1} \cdots \frac{x_i + \lambda\tau - c + 1}{n - c + 1} = \frac{\Pi_{j=0}^{c-1}(x_i + \lambda\tau - j)}{\Pi_{j=0}^{c-1}(n-j)} \tag{4.5}$$

According to Equation (4.3) and (4.5), we have:

$P(A) = \Pi_{i=1}^{t}(\frac{\Pi_{j=0}^{c-1}(x_i+\lambda\tau-j)}{\Pi_{j=0}^{c-1}(n-j)}) = \frac{\Pi_{i=1}^{t}(x_i+\lambda\tau)\Pi_{i=1}^{t}(x_i+\lambda\tau-1)\ldots\Pi_{i=1}^{t}(x_i+\lambda\tau-c+1)}{\Pi_{i=1}^{t}\Pi_{j=0}^{c-1}(n-j)}$.

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

$$P(A) = \frac{f_0(x_1, \ldots, x_t) f_1(x_1, \ldots, x_t) \ldots f_{c-1}(x_1, \ldots, x_t)}{\Pi_{i=1}^{t}\Pi_{j=0}^{c-1}(n-j)} \tag{4.6}$$

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

$$For\ 0 \leq j \leq c-1 : f_j(x_1, \ldots, x_t) = \Pi_{i=1}^{t}(x_i + \lambda\tau - j) \geq 0 \tag{4.7}$$

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 [81], we have: $(\Pi_{i=1}^{t}(x_i + \lambda\tau))^{\frac{1}{t}} \leq \frac{\sum_{i=1}^{t}(x_i+\lambda\tau)}{t}$, and the equality holds if and only if $x_1 + \lambda\tau = x_2 + \lambda\tau = \ldots = x_t + \lambda\tau$, *i.e.*, if and only if $x_1 = x_2 = \ldots = x_t = n\alpha$ (Equation (4.4)). This is equivalent to $\Pi_{i=1}^{t}(x_i+\lambda\tau) \leq (\frac{\sum_{i=1}^{t}(x_i+\lambda\tau)}{t})^t$, in which the equality holds if and only if $x_1 = x_2 = \ldots = x_t = n\alpha$. Similarly, we have the following inequality for the general case: $f_j(x_1, \ldots, x_t) = \Pi_{i=1}^{t}(x_i + \lambda\tau - j) \leq (\frac{\sum_{i=1}^{t}(x_i+\lambda\tau-j)}{t})^t$, where $0 \leq j \leq c-1$, and the equality holds if and only if $x_1 + \lambda\tau - j = x_2 + \lambda\tau - j = \ldots = x_t + \lambda\tau - j$, *i.e.*, the condition for maximizing $f_j(x_1, \ldots, x_t)$ is:

$$For\ 0 \leq j \leq c-1 : f_j(x_1, \ldots, x_t) = max(f_j(x_1, \ldots, x_t)),\ if\ and\ only\ if\ x_1 = \ldots = x_t = n\alpha \tag{4.8}$$

According to Equation (4.6), (4.7) and (4.8), and due to $\Pi_{i=1}^{t}\Pi_{j=0}^{c-1}(n-j) > 0$, $P(A) = max(P(A))$ if and only if $x_1 = x_2 = \ldots = x_t = n\alpha$, *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.

□

4.4 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.

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. We make use of two pseudo-random functions (PRFs) h and γ with the following parameters:

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

RDC-SR **overview.** Like any RDC system for a multiple-server setting [21–23, 28], 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 [22], 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 4.3.2). The client then generates verification tags for every replica, one tag per file block. Each verification tag is computed similarly as in [10], namely as a message authentication code by combining universal hashing with a PRF [59–62]. After having generated t distinct replicas and the corresponding verification tags, the client sends those replicas to t different data centers of the CSP (one replica per server), and the set of all verification tags to each data center. The client also makes public the key used for generating the distinct replicas, so that the servers can use it during Repair to generate new replicas on their own.

During the Challenge phase the client acting as the verifier 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 [8], 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 4.2.2): 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 (Section 4.3.2), 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 (refer to Table C.1) and permit the servers to collaborate among themselves to restore the corrupted replica (the key for generating distinct replicas is known to the CSP). Thus, the system only imposes a small management load on the client (data owner).

A detailed description of RDC-SR is provided in Figures 4.2 and 4.3, together with the following explanation of the three phases.

The Setup **phase.** The client first generates keys K_1 and K_2. K_1 will be used to compute the verification tags and K_2 will be used in generating distinct replicas. It then picks s random numbers, η, and threshold τ (refer to Section 4.5 – Parameterization and Guidelines on how to exactly determine η and τ). The client then calls GenReplicaAndMetadata t times in order to generate t distinct replicas and the corresponding verification tags. Each replica will be sent to a server located in a different data center of the CSP. The entire set of verification tags will be sent to each server. The client may then delete the original file and only keep a small amount of key material.

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

The Challenge **phase.** For this phase, we integrate spot checking [8–10] with our new model introduced in Section 4.2.2. The client (verifier) simultaneously sends a challenge request to each of the t servers. For each challenge, the client selects c random replica

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

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

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

The client may now delete the file $\mathtt{F}$ and stores only a small, constant, amount of data: K_1, $\delta_1, \ldots, \delta_s$, η, 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 $(\rho_1, \ldots, \rho_s, \mathtt{t}) \leftarrow \mathsf{GenProof}(Q, \mathtt{F}_i, \mathtt{t}_{i1}, \ldots, \mathtt{t}_{in})$
3. S_i sends to C the proof of possession $(\rho_1, \ldots, \rho_s, \mathtt{t})$
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 $(\rho_1, \ldots, \rho_s, \mathtt{t})$ by running $\mathsf{CheckProof}(K_1, \delta_1, \ldots, \delta_s, Q, \rho_1, \ldots, \rho_s, \mathtt{t}, i)$

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

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

Figure 4.2 RDC-SR: a replication-based RDC system with Server-side Repair.

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.

KeyGen(1^κ): 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 $1 \leq j \leq n$:
   - Mask block $\mathtt{b}_j$ at the symbol level and get $\mathtt{m}_{ij}$:
     For $1 \leq k \leq s$: $\mathtt{m}_{ijk} = \mathtt{b}_{jk} + \sum_{l=1}^{\eta} \gamma_{K_2}(i||j||k||l)$
3. Compute verification tags:
   For $1 \leq j \leq n$: $\mathtt{t}_{ij} = h_{K_1}(i||j) + \sum_{k=1}^{s} \delta_k \mathtt{m}_{ijk}$
4. Return $(\mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}, \mathtt{F}_i = \{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\})$

GenProof($Q, \mathtt{F}_i, \mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}$):

1. Parse Q as a set of c pairs (j, v_j). Parse $\mathtt{F}_i$ as $\{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\}$.
2. Compute ρ and $\mathtt{t}$:
   - For $1 \leq k \leq s$: $\rho_k = \sum_{(j,v_j) \in Q} v_j \mathtt{m}_{ijk} \; mod \; p$
   - $\mathtt{t} = \sum_{(j,v_j) \in Q} v_j \mathtt{t}_{ij} \; mod \; p$
3. Return $(\rho_1, \ldots, \rho_s, \mathtt{t})$

CheckProof($K_1, \delta_1, \ldots, \delta_s, Q, \rho_1, \ldots, \rho_s, \mathtt{t}, i$):

1. Parse Q as a set of c pairs (j, v_j)
2. If $\mathtt{t} = \sum_{(j,v_j) \in Q} v_j h_{K_1}(i||j) + \sum_{k=1}^{s} \delta_k \rho_k \bmod p$, return "success". Otherwise return "failure".

Figure 4.3 Components of RDC-SR.

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 [21–23]. The client contacts the CSP, reports the corruption, and coordinates the CSP's servers to repair the corruption. The server which is found faulty in the Challenge phase should be replaced by a new server from the same data center. The new server contacts one of the healthy servers, retrieves a replica, un-masks it to restore the original file, and masks the original file to regenerate the corrupted replica. The new server directly retrieves the entire set of verification tags from this healthy server (recall that the entire set of verification tags is stored at every server). Note that the size of the set of all verification tags is always small compared to the data.

A concrete example of using RDC-SR**.** For a concrete example of using RDC-SR, we consider a $4GB$ file F which has $n = 100,000$ $40KB$ 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 an 80-bit security parameter κ, and randomly picks two keys K_1 and K_2 from $GF(2^\kappa)$. 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 4.5. During Setup, C generates 10 replicas $\mathtt{F}_1, \ldots, \mathtt{F}_{10}$ by masking the original file 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 $10MB$ in each of the 10 data centers. C stores locally $K_1, \delta_1, \ldots, \delta_{4000}, \eta$ and τ, which require about $40KB$ 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 [8]. Each challenge request contains the 460-element set $\{(j, v_j) | j \in \mathbb{Z} \wedge 1 \leq j \leq 460\}$, which totals around $5KB$ (j is an integer smaller than $100,000$, v_j is 10 $bytes$). Each server's response contains values $\rho_1, \ldots, \rho_{4000}, \mathtt{t}$, which total around $40KB$ (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.

4.5 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 provide 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 4.3.2, 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, 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.*

Estimating η. From Section 6.4, we have $T_i \leq min(t_i) + min(t_{ij}) + min(t_j) + min(t_R) - 2 * max(t_i)$, which can be further simplified as $T_i \leq t_{ij} + t_j - t_i + t_R$. Let x be the time each of the c challenged file blocks contributes to the generation of the proof by the server. 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 4.2.1), we have $c \cdot x \leq T_i$. Based on the triangle inequality, we always have $t_{ij} + t_j - t_i \geq 0$. To have a coarse evaluation of η, we neglect $t_{ij} + t_j - t_i$, which is always small compared to t_R (milliseconds compared to seconds, as shown in Table 4.1, which contains some typical values based on our experiments for Amazon S3). Thus, we get $c \cdot x \leq t_R$.

Table 4.1 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	0.098
N. California	Virginia	0.08
N. California	Oregon	0.022
Oregon	Virginia	0.098
Oregon	N. California	0.022

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). 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 $t_R = (1 - \alpha) \cdot c \cdot s \cdot \eta \cdot t_{prf}$. We thus get $c \cdot x \leq t_R = (1 - \alpha) \cdot c \cdot s \cdot \eta \cdot t_{prf}$, which means that $\eta \geq \frac{x}{(1-\alpha) \cdot s \cdot t_{prf}}$ (recall that s is the number of

symbols in a file block). The client should choose η as the smallest integer which satisfies this condition.

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

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

4.6 Security Analysis

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 [8, 9]. 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).

As opposed to previous work on RDC, the paradigm we introduce in this paper allows the servers themselves to generate new replicas for repair purposes. This opens the door to a new attack, the *replicate on the fly (ROTF) attack*, in which the economically-motivated servers claim to store t replicas, but in reality they store less than $t|\mathbb{F}|$ data and generate the missing data on the fly upon being challenged by the client. Theorem 4.6.2 shows that RDC-SR can mitigate the ROTF attack executed by an α-cheating adversary (defined in Section 4.3.2):

Lemma 4.6.1. *In* RDC-SR*, by choosing the parameters η and τ according to the guidelines in Section 4.5, 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 4.5, $\eta \geq \frac{x}{(1-\alpha)\cdot s \cdot t_{prf}}$, *i.e.*, $\eta \cdot s \cdot t_{prf} \cdot (1-\alpha) \cdot c \geq c \cdot x$. Since τ is computed as $c \cdot x + 2 \cdot t_i$ (Section 4.5), which is approximately $c \cdot x$, considering t_i is negligibly small compared to $c \cdot x$. Thus, $\eta \cdot s \cdot t_{prf} \cdot (1-\alpha) \cdot c \geq \tau$, *i.e.*, to generate the $(1-\alpha)c$ missing blocks, the cheating server at least needs $\eta \cdot s \cdot t_{prf} \cdot (1-\alpha) \cdot c$ computation, which cannot be done within τ based on her own computational power. □

Theorem 4.6.2. *In* RDC-SR*, by choosing the parameters η and τ according to the guidelines in Section 4.5, an α-cheating adversary can successfully execute the ROTF attack without being detected with a probability of at most $\alpha^c c(1-\alpha)$, where c is the number of file blocks checked by the client in a challenge.*

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

Proof. Per Definition 4.3.1, an α-cheating adversary is an economically-motivated adversary that only uses $\alpha t|\mathtt{F}|$ storage (where $1/t \leq \alpha \leq 1$). We have established in Section 4.5 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 4.5, 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 4.6.1, if we choose the parameters η and τ according to the guidelines in Section 4.5, 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: $\binom{n\alpha}{c} + \binom{n\alpha}{c-1} + ... + \binom{n\alpha}{c-c(1-\alpha)+1}$, where n is the total number of file blocks.

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

$$P(E_1) \leq \frac{\binom{n\alpha}{c}c(1-\alpha)}{\binom{n}{c}} = \frac{\binom{n\alpha}{c}}{\binom{n}{c}}c(1-\alpha) = \frac{n\alpha(n\alpha-1)...(n\alpha-c+1)}{n(n-1)...(n-c+1)}c(1-\alpha) = \frac{n\alpha}{n}\frac{n\alpha-1}{n-1}...\frac{n\alpha-c+1}{n-c+1}c(1-\alpha) = \alpha\frac{n}{n}\alpha\frac{n-\frac{1}{\alpha}}{n-1}...\alpha\frac{n-\frac{c-1}{\alpha}}{n-(c-1)}c(1-\alpha) \leq \alpha^c c(1-\alpha).$$

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

4.7 Implementation and Evaluation

4.7.1 Background on Amazon's Cloud Services (AWS)

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 C.1 and C.2 of Appendix C.1, 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).

4.7.2 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 7.5GB Memory, created from Amazon Linux AMI 64-bit image). The client is run on a machine located in our institute, equipped with Intel Core 2 Duo system with two CPUs (each running at 3.0GHz, with a 6144KB cache), 333GHz frontside bus, 4GB RAM and a Hitachi HDP725032GLA360 360GB hard disk with ext4 file system. In the following, our EC2 instances and S3 data are located in the Oregon region, unless noted otherwise. The prototype for RDC-SR has been implemented in C and uses OpenSSL version 1.0.0e [44] for cryptographic operations.

From Section 4.5, we have $\eta \geq \frac{x}{(1-\alpha)\cdot s\cdot t_{prf}}$ and we also choose $x = 0.034\ sec$. We estimate $t_{prf} = 4.3\ \mu sec$ 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 4.5 that once α is fixed, η can be computed). We use these values for α to reflect an economically-motivated CSP (such a CSP would not likely be interested in saving a small amount of storage, so we 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 4.4: 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 [19, 20]).

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 (20MB, 50MB, 80MB, and 100MB). Table 4.2 shows the throughput for total preprocessing and its different components.

We have several observations for Table 4.2: Firstly, the throughput of masking operation decreases when α increases. This is expected because a larger α means that it is more difficult to detect the adversarial behavior, thus, we need a larger η, hence more computations are required for masking. Secondly, the throughput of verification

tag computation is independent of α, due to the fact that the verification tags are computed over the masked replica, which is independent of η, hence independent of α. Thirdly, the throughput of total preprocessing, which includes masking and verification tag computation, is always close to but a little smaller than the throughput of masking, since the verification tag computation is very efficient (can generate verification tags for more than 5MB data in one second) and only has a small impact to the total preprocessing time.

Table 4.2 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

Challenge. The client issues a challenge to the server (run in an EC2 large instance). The server samples blocks from S3 in the same region, and computes and sends back the proof. The client then checks the proof. For simplicity, we only challenge the server running in EC2 Oregon which is responsible for the replica stored in S3 Oregon. The number of blocks to be challenged is $c = 400$, which provides a high guarantee to detect data corruption by the server [8]. For the chosen values of α (Table 4.2) and c, the probability that a server performs the ROTF attack without being detected is less than $1.38 * 10^{-37}$(cf. Section 4.6). Amazon S3 offers a REST API to access data, which is based on the HTTP/1.0 protocol. Although HTTP supports operations on multiple ranges of the target object in one request, Amazon S3 only supports one range. This means that in order to sample 400 random blocks, we must send 400 different requests for a one-block range. This explains partially

the large variation we observe in block access time for S3 (Figures 4.4(b) and 4.5(b)), thus we average the block access time over 100 runs. We examine two cases:

- *Benign case:* The CSP is honest, *i.e.*, it strictly stores the replicas in the corresponding regions according to the contract. Upon challenge, the server uses the data from the same region to pass the challenge. In this case, the total server computation includes sampling challenged blocks from S3 of the same region and computing the proof.
- *Adversarial case:* The CSP is cheating by not storing all replicas in their entirety according to the contract. The malicious CSP adopts the best attack strategy described in Section 4.3.2. 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. Figures 4.4 and 4.5 show the server computation and client computation for both cases.

For the benign case, we observe from Figure 4.4 that the total server computation and its various components as well as the client (verifier) computation are independent of file size and of α. This is expected because: Firstly, we rely on spot checking [8] which always randomly samples a fixed number of blocks from the masked replica, thus can maintain constant server/client computation. Secondly, during a challenge, the operations on both server and client are over the masked replica, which is independent of η, hence independent of α. Figure 4.4(d) shows that the time for the client to check the proof is less than 7 $msec$, which justifies our claim that the system imposes a small load on the verifier during the challenge phase.

For the adversarial case, we observe from Figure 4.5 that the total server computation and its various components are independent of filesize. The reason has been explained in

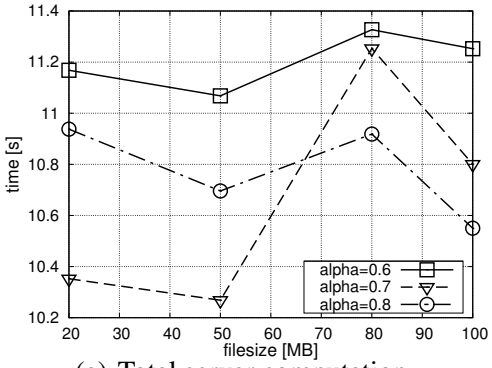
the benign case. For Figure 4.5(c), we expected to see that the masking time is independent of α, because: The malicious server always stores only an α fraction of the corresponding data, and generates the $1-\alpha$ fraction of challenged blocks on the fly (by masking). Larger α means that the malicious server has to generate less challenged blocks but generating one challenged block will be more expensive, thus, the masking time for the $1-\alpha$ fraction of challenged blocks should be almost constant. Figure 4.5(c) shows that for the case of $\alpha = 0.7$, the masking time is larger than those of other two cases. This discrepancy can be explained because we must always choose η as an integer number. The server masking time is $400(1-\alpha) \cdot s \cdot \eta \cdot t_{prf}$, which is determined by the multiplication of $1-\alpha$ and η. For the case of $\alpha = 0.7$, the minimum integer for η is 7, thus, $(1-\alpha) \cdot \eta = 2.1$. For both cases $\alpha = 0.6$ and $\alpha = 0.8$, $(1-\alpha) \cdot \eta = 2 < 2.1$. This explains where such a discrepancy comes from. Note that there is a lower bound on the server masking time, because $400(1-\alpha) \cdot s \cdot \eta \cdot t_{prf} \geq 400(1-\alpha) \cdot s \cdot t_{prf} \cdot \frac{x}{(1-\alpha) \cdot s \cdot t_{prf}} = 400x = 13.6\ sec.$ We observe that most of the points in Figure 4.5(c) are over this lower bound, except the point in 20MB filesize when $\alpha = 0.6$, but we still consider this point as valid since it is only 1% smaller. The existence of the lower bound for the server masking time guarantees that even if the malicious server has the magic power to access the data and compute the proof instantly (*i.e.*, the times shown in Figure 4.5(b) and Figure 4.5(d) are 0), it still cannot cheat successfully, since the time for generating the $1-\alpha$ fraction of challenged blocks will be always larger than $400x$, which is the total server computation for the benign case.

Figure 4.5(d) shows that the time for the server to compute the proof varies with α. We can still conclude that this time is independent of α given that the variance is quite small (around 1%).

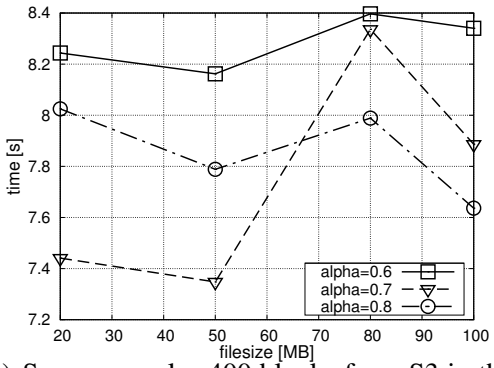
According to the guidelines for establishing the time threshold τ in Section 4.5, τ should be 13.7 sec (c=400, x = 0.034 sec, $t_i = 0.045\ sec$ based on our experiments). We see that 95% of the individual runs for Figure 4.4(a) are below this threshold, and 100% of the individual runs for Figure 4.5(a) are above this threshold. This confirms the practical value of using a time threshold to establish if the CSP is malicious.

Repair. We assume that the replica stored in S3 Oregon has been found corrupted, and the replica stored in S3 California is retrieved to repair the corruption. The repair server runs in a large instance from EC2 Oregon. The server downloads the replica from S3 California and masks it to generate the replica for S3 Oregon. The server also downloads all the verification tags from another S3 region (this time is negligible in our experiment). The results are shown in Figure 4.6 (this includes time for masking to generate a new replica, as described Table 4.2). We observe from Figure 4.6(a) that, for repairing one replica, total server computation increases with α. This is because, as shown in Preprocessing, larger α will result in larger masking computation, and the masking computation dominates the total repair computation.

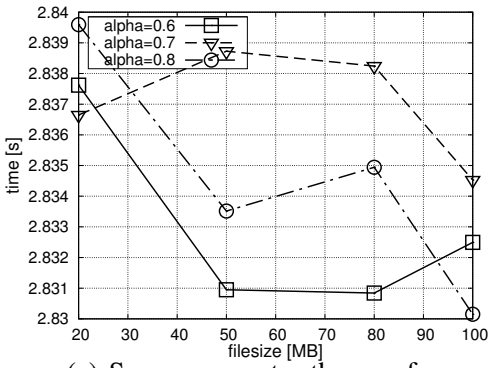
One significant advantage in the repair phase is that the client can be kept lightweight, *e.g.*, the client only needs to exchange a few messages to coordinate the repair procedure. This justifies our claim that the system imposes a small management load on the data owner during repair.



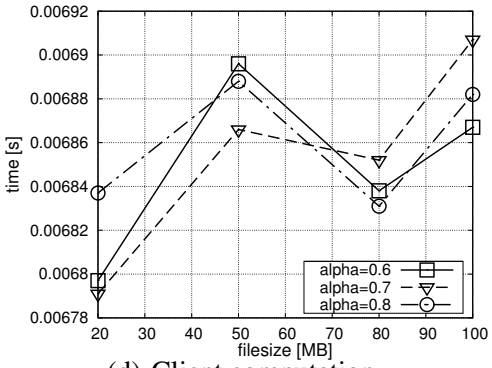
(a) Total server computation.



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

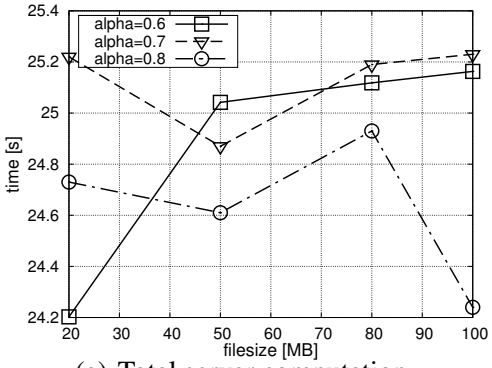


(c) Server computes the proof.

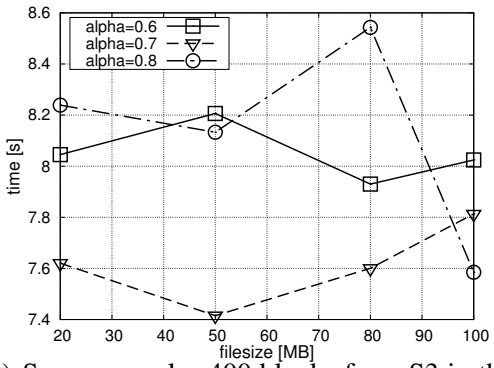


(d) Client computation.

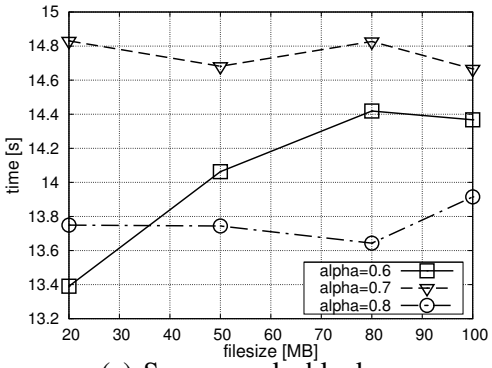
Figure 4.4 Computational cost for both the server and the client in challenge phase (benign case).



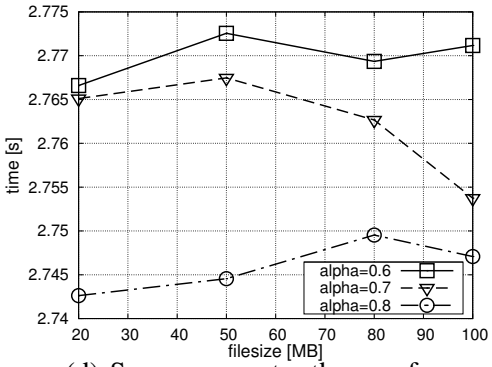
(a) Total server computation.



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

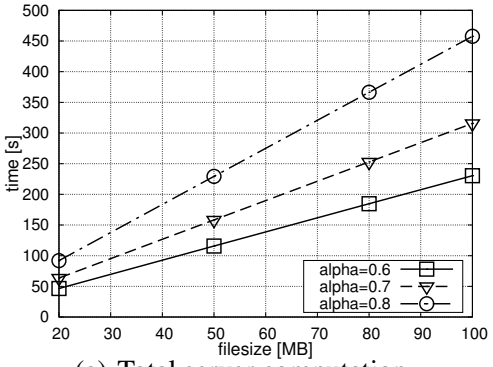


(c) Server masks blocks.

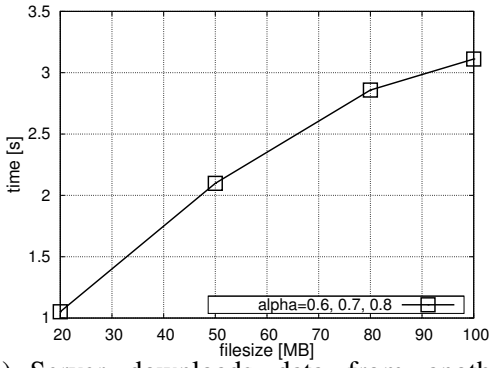


(d) Server computes the proof.

Figure 4.5 Computational cost for the server and its various components in challenge phase (adversarial case).



(a) Total server computation.



(b) Server downloads data from another region.

Figure 4.6 Computational cost for repairing a replica.

CHAPTER 5

AN ENHANCED REMOTE DATA CHECKING SCHEME SUPPORTING SERVER-SIDE REPAIR

In this chapter, an enhanced RDC-SR scheme for replication-based distributed storage systems (ERDC-SR) is introduced. Similar to RDC-SR, ERDC-SR can allow server-side repair and place a minimal load on the data owner. Different than RDC-SR, ERDC-SR relaxes the assumption that the computational power of the CSP does not grow over time, and is thus more suitable for real-world applications.

When a replication-based 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. During Setup, the data owner creates multiple replicas of the file `F`, and stores them at multiple storage servers (one replica at each server). During the Challenge phase, the data owner periodically checks whether all the replicas have remained intact. If a replica is found corrupt during Challenge, the data owner will take actions to Repair it, such that the desired redundancy level in the system is restored. The Challenge and Repair phases will alternate over the lifetime of the system.

Conforming to the 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. RDC-SR (Chapter 4) initiates the investigation of server-side repair, and defends against a new replicate-on-the-fly (ROTF) attack. In an ROTF attack, an untrusted CSP who knows the

secrets on how to generate replicas, will try to convince the data owner that it stores the expected number of replicas by generating on the fly a replica being challenged, even though its actual number of stored replicas is much less than the expected. RDC-SR formalizes the adversary who performs the ROTF attack as an α-cheating adversary, which is an economically motivated untrusted CSP who is cheating by storing an α-fraction of the expected storage.

The basic idea of RDC-SR is as follows: during Setup, the data owner creates multiple replicas, each of which is created from F by using a controllable amount of masking (the masking factor η is determined by the CSP's computational power); during Challenge, the verifier challenges each server to obtain a proof of data possession and checks whether the response time is within a pre-determined threshold τ. RDC-SR can successfully defend against an α-cheating CSP who will not grow its computational power over time (*i.e.*, a static α-cheating adversary). In practical applications, the α-cheating CSP may increase its budget to grow its computational power, and RDC-SR cannot defend against this type of dynamic α-cheating adversary because, firstly, in RDC-SR, the time threshold τ is fixed during Setup, *i.e.*, after the Setup phase, if the CSP upgrades its computational power, the α-cheating adversary can easily pass the verification without being detected by performing the ROTF attack; secondly, the client cannot dynamically adjust τ in time after Setup, since it does not have any knowledge on when the CSP will 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 always obtain an assurance of the reliability and fault tolerance of the data, even if the CSP grows its computational power, unbeknownst to the client.

A Straw Man Solution (the RDC-SR-1 scheme). To defend against the dynamic α-cheating CSP, one can simply extend RDC-SR in the following way: 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 pre-processes 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, RDC-SR-1, can successfully defend against the dynamic α-cheating adversary. Unfortunately, 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 dependency between each of the replica block and multiple original file blocks.

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. Though a lot in common, they are different in several aspects. Firstly, both schemes allow server-side repair, and at the same time, overcome the ROTF attack performed by the economically-motivated malicious CSP. Yet, 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 who can grow its computational power over time. Secondly, during the Setup phase, both schemes pre-process the original file, generating different replicas and the corresponding metadata. However, to create a replica, RDC-SR

relies on a controllable amount of masking, and ERDC-SR utilizes a variant of butterfly encoding [82]. Thirdly, in RDC-SR, each replica block depends on only one original file block. Oppositely, in ERDC-SR, each replica block depends on multiple original file blocks. Thus, during the Challenge phase, to answer a challenge by performing the ROTF attack, the adversary in ERDC-SR needs 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, and ERDC-SR can only be applied to files that are large enough (*e.g.*, under typical parameters, a file needs to be at least $60MB$).

5.1 System and Adversarial Model

5.1.1 System Model

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

5.1.2 Adversarial Model

We assume the CSP is rational and economically motivated, *i.e.*, it will try to cheat only if cheating cannot be detected and meanwhile, can achieve certain economic benefits, such as using less storage than required by contract.

Similar to RDC-SR, we consider an α-cheating adversary, which is an economically-motivated adversary that can successfully pass a challenge by only storing an α fraction of the expected storage (where $\frac{1}{t} \leq \alpha \leq 1$). Different from RDC-SR, in this work, we consider a **dynamic α-cheating adversary**, a more powerful adversary who can grow its computational power over time. A dynamic α-cheating adversary refers to an α-cheating CSP, who initially has a known amount of computational power, and is capable of growing its computational power at a known rate over time. Compared to the static α-cheating adversary considered in RDC-SR, this captures a more flexible and realistic setting, in which a CSP may choose to increase its budget on the computational power due to its business strategy. Note that it is reasonable to assume the CSP grows its computational power with a fixed growth rate known by the client since, firstly, a rational and economically motivated CSP will increase its budget steadily, rather than arbitrarily, and the client can possibly estimate how the CSP will grow its budget on the computational power from the CSP's historical data; secondly, the unit price of the computational power usually decreases steadily, and the client can estimate how this price will decrease based on the historical records, *e.g.*, the historical price of processor and memory.

5.2 An Enhanced RDC Scheme with Server-side Repair

In this section, we propose ERDC-SR, an Enhanced RDC scheme which can support Server-side Repair, and can defend against a dynamic α-cheating adversary. The basic idea

of ERDC-SR is two-fold: firstly, 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 knowledge of the CSP's initial computational power and the growth rate. This can ensure that a cheating CSP is not able to compute the intermediate blocks and hence the challenged blocks within τ when answering a challenge.

In ERDC-SR, we make each replica block depend on a subset of β original file blocks. We term β as dependency factor. A butterfly network [82] was previously proposed to create dependency between an encoded block and all the original blocks in a file. By tuning this solution, we can create dependency between an encoded block and arbitrary number of original file blocks. In the following, we first design a β-butterfly encoding, and then propose an ERDC-SR scheme based on β-butterfly encoding. In Figure 5.1, we provide a reference sheet showing various parameters of ERDC-SR used in this section.

- n: the number of PDP blocks in a file $\mathtt{F}$.
- c: the number of PDP 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 duing 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.

Figure 5.1 A reference sheet for various parameters.

5.2.1 β-butterfly Encoding

As shown in Figure 5.2, when creating 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 $1 \leq j \leq log\beta$. The cryptographic transformation used in Figure 5.2 has the following properties: firstly, each bit of the pair of output blocks depends on each bit of the pair of input blocks; secondly, each output block has the same size as the input block. In Section 5.3.1, we provide an instantiation for the cryptographic transformation used in ERDC-SR. Compared to the butterfly encoding used in [82], in which each of the resulting blocks depends on all the original file blocks, the β-butterfly encoding offers more flexibility: each resulting block (in level $log\beta$) depends on β original file blocks (*i.e.*, blocks on level 0), where $1 \leq \beta \leq n$. β determines the amount of dependency introduced between the replica blocks and the original file blocks. By having a suitable β value (Section 5.3.2), we can ensure 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.

5.2.2 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

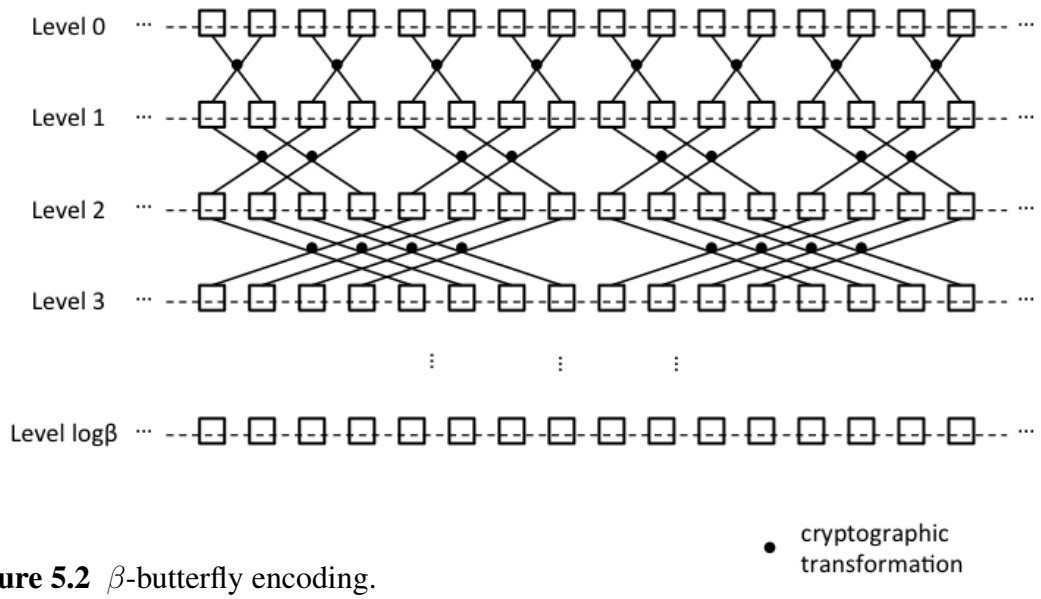


Figure 5.2 β-butterfly encoding.

the index of a symbol in a block (*i.e.*, $k \in \{1 \dots s\}$). Let κ be a security parameter. Let B denote the blocksize. Let h and f be two PRFs, E be a cryptographic transformation (Section 5.3.1) and D be the reverse operation of the cryptographic transformation. f, E and D have the following parameters:

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

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_1 and the replica index, and applies the β-butterfly encoding to the original file with this key, which will be used in each cryptographic transformation during β-butterfly encoding. For each replica, the client computes a set of verification tags based on key 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 Figures 5.3 and 5.4, together with the following explanation.

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 according to the guidelines in Section 5.3. It then computes the dependency factor β (Section 5.3), and calls EnhancedGenReplicaAndMetadata 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 EnhancedGenReplicaAndMetadata, 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 (Figure 5.2), 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 (as the corrupted server), which will be used to replace the corrupted server, contacts one of the healthy servers, retrieves a replica, decodes (*i.e.*, ButterflyDecode, which is actually the reverse operation of ButterflyEncode) it to restore

We construct ERDC-SR in three phases, Setup, Challenge and Repair by utilizing the components in both Figures 4.3 and 5.4. 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 5.3
2. Compute and outsource the replicas and the verification tags
   For $1 \leq i \leq t$:
   - Run $(\mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}, \mathtt{F}_i) \leftarrow \mathsf{EnhancedGenReplicaAndMetadata}(\beta, K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s)$
   - 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 $\mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}$ 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 (Chapter 4).

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 $(\mathtt{F}_y) \leftarrow \mathsf{ButterflyEncode}(\beta, K_2, \mathtt{F}, y)$

Figure 5.3 ERDC-SR: an Enhanced replication-based RDC system with Server-side Repair.

the original file, and encodes (*i.e.*, ButterflyEncode) the original file to regenerate the corrupted replica. The new server directly retrieves from this healthy server the entire set of verification tags.

5.3 Guidelines for ERDC-SR

In this section, we provide guidelines on applying ERDC-SR in practical applications. 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

EnhancedGenReplicaAndMetadata$(\beta, K_1, K_2, \mathtt{F}, i, \delta_1, \ldots, \delta_s)$:

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 $1 \leq j \leq n$: $\mathtt{t}_{ij} = h_{K_1}(i||j) + \sum_{k=1}^{s} \delta_k \mathtt{m}_{ijk}$
4. Return $(\mathtt{t}_{i1}, \ldots, \mathtt{t}_{in}, \mathtt{F}_i = \{\mathtt{m}_{i1}, \ldots, \mathtt{m}_{in}\})$

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 $(\mathtt{F} = \{\mathtt{b}_1 = G[0], \mathtt{b}_2 = G[1], \ldots, \mathtt{b}_n = G[n-1]\})$

Figure 5.4 Components for ERDC-SR.

ERDC-SR scheme. For convenience, we provide a reference sheet in Figure 5.5 for all the parameters used in ERDC-SR.

• n: the number of PDP blocks in a file $\mathtt{F}$.
• m: the number of 64-bit blocks in a PDP block.
• c: the number of PDP 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 duing 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.
• ρ: the annual growth rate of CSP's computational power.
• ϕ: the time period (*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: computational time needed for one cryptographic transformation based on CSP's computational power at Setup.
• u: the computational time needed for one AES operation over a 128-bit block based on CSP's computational power at Setup.

Figure 5.5 A reference sheet for all the parameters used in ERDC-SR.

5.3.1 Instantiating The Cryptographic Transformation

The cryptographic transformation E used in ERDC-SR has the properties 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 (*e.g.*, AES) cannot work, because there is a gap between the blocksize used for block cipher purpose and that used for provable data possession (PDP [8]) purpose: block ciphers like AES always use a small blocksize, *e.g.*, AES uses 128 bits as the blocksize; oppositely, the blocksize used in ERDC-SR for PDP purpose should be always large enough (*e.g.*, $4KB$), such that the storage overhead for the verification tags remains reasonable. It may seem that this gap can be filled by using a block cipher with a mode of operation such as CBC [83] which, however, is not sufficient, since it may not necessarily guarantee the property that each bit of the output depends on each bit of the input. In the following, we provide a proper instantiation for E, which can be achieved in two steps:

1. View the input (*i.e.*, two PDP blocks) of E as a collection of $2 \cdot m$ 64-bit blocks.

2. Apply a butterfly encoding (Figure 5.2) 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 $1 \leq j \leq log(2 \cdot m)$.

This full butterfly encoding can achieve "strong mixing" [82], such that each bit of the output (two PDP blocks) depends on each bit of the input (two PDP blocks). In Figure 5.6, we show a concrete example for the instantiation of a cryptographic transformation where each PDP block has 4 64-bit blocks. Correspondingly, D (Section 5.2) can be instantiated as the reverse process of E. Note that we should always choose m as a power of 2, so that $log(2 \cdot m)$ is always a positive integer.

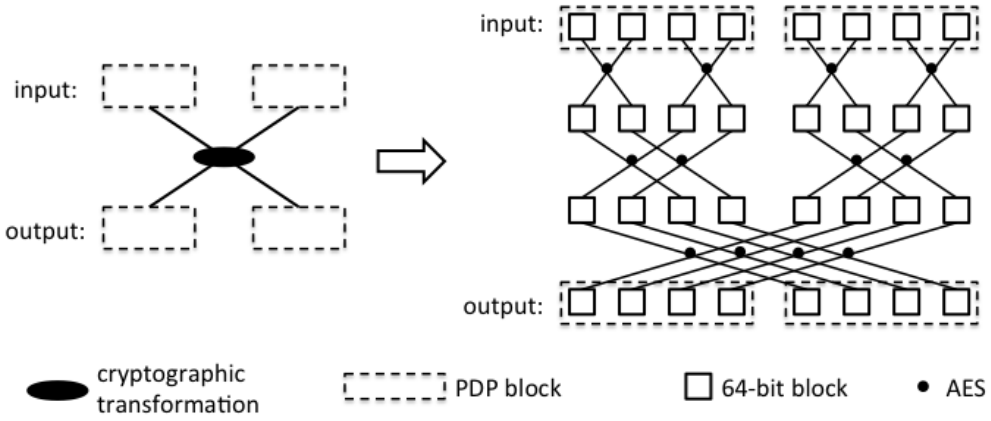


Figure 5.6 An example for the instantiation of cryptographic transformation.

5.3.2 Estimating The Parameters

In the following, we first provide the best adversarial storage strategy for ERDC-SR, and then provide guidelines on how to estimate the parameters used in ERDC-SR such that one can apply ERDC-SR in practical applications. Among the parameters shown in Figure 5.5, α, ρ and ϕ are known, and c can be determined according to PDP ([8]). 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 (Chapter 4), the best data distribution strategy for an α-cheating CSP is to store in each storage server $n\alpha$ blocks, which can be the original file blocks, intermediate blocks (Figure 5.2), blocks of the corresponding replica (replica blocks), or a combination of different types of blocks. According to Appendix D.2, if the adversary only stores a block from level i ($0 \leq i \leq log\beta$) of the butterfly network (Figure 5.2) and is able to access the original file, 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.

Estimate τ. Similar to RDC-SR (Chapter 4), the time threshold τ in ERDC-SR can be computed as $c \cdot x + 2 \cdot t_i$, where 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, and t_i denotes the network delay between the challenged server and the client.

Estimate m, n and e. Considering $|F|$ is the size of a replica in bytes, n (the number of PDP blocks in a replica) and m (the number of 64-bit blocks in a PDP block) have the following relation: $n \cdot m \cdot 8 = |F|$, *i.e.*, $n = \frac{|F|}{8 \cdot m}$. Each replica should have at least 2 PDP blocks, *i.e.*, $n \geq 2$. Thus, $\frac{|F|}{8 \cdot m} \geq 2$. From Section 5.3.1, we have $e = \frac{2 \cdot m}{2} \cdot log(2 \cdot m) \cdot u$, because E is instantiated as a butterfly encoding over 2 PDP blocks (*i.e.*, $2 \cdot m$ 64-bit blocks) with $log(2 \cdot m)$ levels, and u is the computational time needed for one AES

operation over a 128-bit block during Setup (see Figure 5.5). According to Appendix D.3, $e > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{n+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$, *i.e.*, $\frac{2\cdot m}{2} \cdot log(2 \cdot m) \cdot u > \frac{((1+\rho)^{\phi}-1)\cdot\tau}{n+\frac{1}{2}\cdot c\cdot(1-\alpha)-2}$. 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 conditions can be satisfied simultaneously: First, $\frac{|F|}{8\cdot m} \geq 2$; Second, $log(2\cdot m)\cdot u\cdot\frac{|F|}{8}+m\cdot log(2\cdot m)\cdot u\cdot(\frac{1}{2}\cdot c\cdot(1-\alpha)-2) > ((1+\rho)^{\phi}-1)\cdot\tau$; Third, 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 can not find such an integer which can satisfy the aforementioned conditions, it will output -1, *i.e.*, such an m_{min} value does not exist. This may happen when $|F|$ is small. For a concrete example, when $|F| = 100MB$, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12sec$, and $u = 0.1\mu s$ (estimated from our local machine, which is equipped with Intel Core $i5-4250U$ processor and $4GB$ RAM), m_{min} is 64 (*i.e.*, the minimum size of a PDP block is $512B$). After having determined the minimum value of m, how to pick 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 PDP blocksize, which will lead to less storage overhead for verification tags, but more computation and communication overhead in each challenge. In general, we choose m such that the PDP blocksize is from KBs (*i.e.*, $4KB$) to tens of KBs (*i.e.*, $40KB$). In other words, to guarantee we can always choose a meaningful PDP blocksize, m_{min} should be small enough. We provide in Table 5.1, 5.2, 5.3 and 5.4 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 , *e.g.*, in Table 5.1, the filesize should be at least $60MB$ in order to choose m as 2048.

Algorithm 1: Compute m_{min}

input : $|F|$, c, α, u, ρ, ϕ and τ
output: The minimum value of m
i=0;
m=1;
while $\frac{|F|}{8 \cdot m} \geq 2$ *and*
$log(2 \cdot m) \cdot u \cdot \frac{|F|}{8} + m \cdot log(2 \cdot m) \cdot u \cdot (\frac{1}{2} \cdot c \cdot (1 - \alpha) - 2) \leq ((1 + \rho)^{\phi} - 1) \cdot \tau$ **do**
    ++i;
    m=2^i;
end
if $\frac{|F|}{8 \cdot m} < 2$ **then**
    m=-1;
end
return m

Table 5.1 m_{min} When $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12sec$, and $u = 0.1\mu s$

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 5.2 m_{min} When $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 5$, $\tau = 12sec$, and $u = 0.1\mu s$

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 5.3 m_{min} When $c = 460$, $\alpha = 0.9$, $\rho = 40\%$, $\phi = 2$, $\tau = 12sec$, and $u = 0.1\mu s$

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 5.4 m_{min} When $c = 460$, $\alpha = 0.9$, $\rho = 40\%$, $\phi = 5$, $\tau = 12sec$, and $u = 0.1\mu s$

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

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 PDP block is $4KB$), which is larger than the minimum value 64. Thus, n will be $25,600$. In general, we choose m such that, firstly, it is larger than or equal to m_{min}; secondly, a PDP block should have a suitable size, *e.g.*, 4KB.

Once m is fixed, we can compute e by $\frac{2 \cdot m}{2} \cdot log(2 \cdot m) \cdot u$. Using the aforementioned example, in which m is chosen as 512, we can compute e as $512\mu s$.

Estimate 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, considering each cryptographic transformation is expensive enough (Appendix D.3). 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 under the condition that the 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 D.5, when the condition $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$ 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 $p = \Pi_{i=1}^{c \cdot (1-\alpha)-1} \frac{n-i\beta}{n-i}$. At the end of ϕ, the CSP's computational power will be upgraded 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 $n, c, \alpha, e, \rho, \phi$ and τ. Specifically, β should be chosen such that the following conditions can be satisfied simultaneously: First, $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$; 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 can not 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, the integer to be tested will be doubled. The computational complexity for Algorithm 2 is thus $O(logn)$. In Algorithm 2, in order to determine β, we need to first fix the input parameters $n, c, \alpha, e, \rho, \phi$ and τ, which can be computed as follows:

1. Pick α, ρ and ϕ, which are known in practical applications; pick c according to PDP [8].
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 $|F| = 100MB$, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12sec$, $m = 512$, $n = 25,600$, and $e = 512\mu s$, 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 $|F| = 500MB$, $c = 460$, $\alpha = 0.9$, $\rho = 30\%$, $\phi = 2$, $\tau = 12sec$, $m = 512$ (by applying Algorithm 1, the minimum m will be 2 for this case, and we choose m as 512 to maintain the PDP blocksize as $4KB$), $n = 128,000$, and $e = 512\mu s$, the output of Algorithm 2 will be 1024, *i.e.*, each of the replica blocks should depend on 1024 original file blocks.

Algorithm 2: Estimate β

input : $n, c, \alpha, e, \rho, \phi, \tau$
output: β
$j = 0$;
$\beta = 1$;
$p = \Pi_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i}$;
while $2 \cdot c \cdot (1-\alpha) \cdot \beta \leq n$ *and* $\frac{\frac{1+p}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)\cdot e}{(1+\rho)^{\phi}-1} \leq \tau$ **do**
  $++j$;
  β=2^j;
  $p = \Pi_{i=1}^{c\cdot(1-\alpha)-1} \frac{n-i\beta}{n-i}$;
end
if $2 \cdot c \cdot (1-\alpha) \cdot \beta > n$ **then**
  β=n;
end
return β

5.4 Security Analysis for ERDC-SR

According to Section 5.3.2, 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 of 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 5.4.1 and Theorem 5.4.2 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 5.3.2, and then retrieves the original file, preprocesses 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 5.4.1. *In* ERDC-SR*, at ϕ_{end}, a cheating server who is storing an α-fraction of the corresponding replica, based on its own computational power, cannot generate the $(1-\alpha)c$ missing blocks and compute a proof to answer a challenge within τ, where c is the number of file blocks checked by the client in a challenge, and the set of parameters τ and β is computed according to the guidelines in Section 5.3.2.*

Proof. According to Section 5.3.2, β is chosen such 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. The computation of the cheating server, based on its own computational power at ϕ_{end}, is at least $P(E) \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot \frac{e}{(1+\rho)^{\phi}}$ for the

former component, and $\frac{c \cdot x}{(1+\rho)^{\phi}}$ for the latter component. Since τ is approximately $c \cdot x$, the computation of the cheating server is 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. □

Theorem 5.4.2. *By choosing the set of β and τ parameters according to the guidelines in Section 5.3.2, at ϕ_{end}, an α-cheating adversary can successfully execute the ROTF attack without being detected with a probability of at most $\alpha^c c(1-\alpha)$, where c is the number of file blocks checked by the client in a challenge.*

Proof. We have established in Section 5.3.2 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 5.3.2, 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 5.4.1, if we choose the parameters β and τ according to Section 5.3.2, 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 4.6.2, $P(V_1) \leq \alpha^c c(1-\alpha)$, thus, $P(V) \leq \alpha^c c(1-\alpha)$. □

5.5 Performance Analysis for ERDC-SR

In this section, we provide an analytical performance analysis for ERDC-SR by comparing ERDC-SR to RDC-SR-1 (a simple extension of RDC-SR presented at the beginning of this chapter) and RDC-SR. 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 three schemes. We further compare the computational time of these three schemes in both the Setup and the Repair phase. Note that throughout this section, we *evaluate the computational time for both* Setup *and* Repair *phase of different schemes in terms of the number of elementary cryptographic operations*, *i.e.*, PRF for both RDC-SR and RDC-SR-1, cryptographic transformation (Section 5.3.1) for ERDC-SR. Let t_{prf} be the computational time required for one PRF during Setup (in

both RDC-SR and RDC-SR-1). e is defined in Section 5.3 as the computational time needed for one cryptographic transformation during Setup (for 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 our instantiation of cryptographic transformation (Section 5.3.1), e can be computed as $m \cdot log(2 \cdot m) \cdot u$, where m is the number of 64-bit blocks in a PDP block and u is the computational time needed for one AES operation over a 128-bit block based on CSP's computational power during Setup (estimated as $0.1\mu s$ in Section 5.3.2). Thus, we need to first estimate m and then estimate e. According to Section 5.3.2, after having computed m_{min}, we choose m such that, (a) $m \geq m_{min}$, and (b) the PDP blocksize is from KBs (*i.e.*, $4KB$) to tens of KBs (*i.e.*, $40KB$), 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 (Chapter 4), we use $40KB$ as the PDP blocksize. For consistency, we choose m as 4096 such that the PDP blocksize in ERDC-SR is close to $40KB$. Correspondingly, e is $5325\mu s$.

For RDC-SR, we use η_0 to denote the masking factor. Thus, the computational time needed to create a new replica from the original file during Setup is approximately $n \cdot s \cdot \eta_0 \cdot t_{prf}$ (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_0 random values generated by the PRF). By estimating η_0 as $\frac{x}{(1-\alpha) \cdot s \cdot t_{prf}}$ (Chapter 4), $n \cdot s \cdot \eta_0 \cdot t_{prf} = \frac{n \cdot x}{1-\alpha} = \frac{n \cdot c \cdot x}{c \cdot (1-\alpha)} \approx \frac{n \cdot \tau}{c \cdot (1-\alpha)}$.

To evaluate the computational time for creating a new replica in RDC-SR-1 from the original file during Setup, we first estimate the parameters τ and η used in RDC-SR-1. 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 $c \cdot (1-\alpha) \cdot s \cdot \eta \cdot \frac{t_{prf}}{(1+\rho)^{\phi}}$ (generating one replica block requires $s \cdot \eta \cdot t_{prf}$ 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: $c \cdot (1-\alpha) \cdot s \cdot \eta \cdot \frac{t_{prf}}{(1+\rho)^{\phi}} + \frac{\tau}{(1+\rho)^{\phi}} > \tau$, *i.e.*, $\eta > \frac{((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha) \cdot s \cdot t_{prf}}$. The computational time needed to create a new replica from the original file during Setup is $n \cdot s \cdot \eta \cdot t_{prf}$. By estimating η as $\frac{((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha) \cdot s \cdot t_{prf}}$, it becomes $\frac{n \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha)}$.

For ERDC-SR, the computational 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 5.3.2 will return a β value for which we can distinguish two cases: $\beta = n$ and $\beta < n$.

- When $\beta = n$, the computational time for creating a distinct replica from the original file during Setup (*i.e.*, $\frac{n}{2} \cdot log\beta \cdot e$) becomes $\frac{n}{2} \cdot logn \cdot e$.
- When $\beta < n$, we approximate the β value in the following way: Recall from Section 5.3.2, $\frac{\frac{1+p}{2} \cdot c \cdot (1-\alpha) \cdot (\beta-1) \cdot e}{(1+\rho)^{\phi}-1} > \tau$. 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 $p = \Pi_{i=1}^{c \cdot (1-\alpha)-1} \frac{n-i\beta}{n-i} \geq 0$, 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 computational time for creating a distinct replica from the original file during Setup (*i.e.*, $\frac{n}{2} \cdot log\beta \cdot e$) 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 RDC-SR-1 **and** RDC-SR**.** In the following, we compare RDC-SR-1 and RDC-SR in both the Setup and the Repair phase. During Setup, the overall workload

for both RDC-SR-1 and RDC-SR contains two components, creating t different replicas and computing t sets of verification tags. Since replica creation is always a lot more expensive than verification tag generation (confirmed by the experimental evaluation for RDC-SR in Chapter 4), we can estimate the overall computational time by simply calculating the time needed for creating t different replicas. Thus, the overall computational time in Setup phase is approximately $\frac{t \cdot n \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha)}$ for RDC-SR-1 and $\frac{t \cdot n \cdot \tau}{c \cdot (1-\alpha)}$ for RDC-SR. We use r_1 to denote the ratio between the overall computational time of RDC-SR-1 and that of RDC-SR during Setup. r_1 is $\frac{\frac{t \cdot n \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha)}}{\frac{t \cdot n \cdot \tau}{c \cdot (1-\alpha)}}$, which can be further reduced to $(1+\rho)^{\phi} - 1$. During Repair, for both RDC-SR-1 and RDC-SR, the overall workload contains two main components, unmasking a replica to generate the original file and masking the original file to generate the corresponding replica. Since unmasking is the reverse operation of masking, the ratio of the overall computational time in Repair between RDC-SR-1 and RDC-SR is equal to r_1. In Table 5.5, we show some concrete values for r_1 by varying ϕ and ρ.

Table 5.5 Concrete Values for r_1 by Varying ϕ and ρ (Recall That r_1 Is The Ratio between The Overall Computational Time of RDC-SR-1 and That of RDC-SR)

ϕ \ ρ	30%	40%
5	2.71	4.38
10	12.79	27.93
15	50.19	154.57

Comparison between ERDC-SR and RDC-SR-1. In the following, we compare ERDC-SR and RDC-SR-1 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 $t \cdot \frac{n}{2} \cdot logn \cdot e$ 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_2 to denote the ratio between the overall computational time of ERDC-SR and that of RDC-SR-1 during Setup.

- When $\beta = n$, r_2 is $\frac{t \cdot \frac{n}{2} \cdot logn \cdot e}{t \cdot \frac{n \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha)}}$, which can be further reduced to $\frac{logn \cdot e \cdot c \cdot (1-\alpha)}{2 \cdot ((1+\rho)^{\phi}-1) \cdot \tau}$. To show some concrete values for r_2, we choose $c = 460$, $\alpha = 0.9$, $\tau = 12s$, $e = 5325\mu s$, $n = 25,600$, and vary ϕ and ρ (see Table 5.6). Note that in the aforementioned examples, each set of $n, c, \alpha, e, \rho, \phi, \tau$ values can guarantee β should be chosen as n (Algorithm 2 in Section 5.3.2). We observe from Table 5.6 that r_2 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 RDC-SR-1. Specifically for some case in Table 5.6 (*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 RDC-SR-1.
- When $\beta < n$, r_2 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_2 always decreases when β increases (note that β is a positive integer which is larger than 1). When $\beta = 64$, $r_2 = \frac{log64}{64-1} = 0.095$. Considering e has an order of magnitude $10^{-3}s$, β (*i.e.*, $\frac{2 \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha) \cdot e} + 1$) should have an order of magnitude 3, which is always larger than 64, *i.e.*, r_2 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 RDC-SR-1.

Table 5.6 Concrete Values for r_2 by Varying ϕ and ρ When $\beta = n$ (Recall That r_2 Is The Ratio between The Overall Computational Time of ERDC-SR and That of RDC-SR-1)

ϕ \ ρ	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 RDC-SR-1, 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 RDC-SR-1 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 RDC-SR-1.

Comparison between ERDC-SR **and** RDC-SR**.** We compare ERDC-SR and RDC-SR in both the Setup and the Repair phase. As mentioned previously, for RDC-SR, the overall computational time in Setup is $\frac{t \cdot n \cdot \tau}{c \cdot (1-\alpha)}$; for ERDC-SR, the overall computational time in Setup is $t \cdot \frac{n}{2} \cdot logn \cdot e$ 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_3 to denote the ratio between the overall computational time of ERDC-SR and that of RDC-SR during Setup.

- When $\beta = n$, r_3 is $\frac{t \cdot \frac{n}{2} \cdot logn \cdot e}{\frac{t \cdot n \cdot \tau}{c \cdot (1-\alpha)}}$, which can be further reduced to $\frac{logn \cdot e \cdot c \cdot (1-\alpha)}{2 \cdot \tau}$. For example, we choose $c = 460$, $\alpha = 0.9$, $\tau = 12s$, $e = 5325\mu s$, and r_3 becomes $0.01 \cdot logn$. In practical applications, n will be always smaller than 2^{100}, thus, for this set of c, α, τ, e parameters, r_3 (*i.e.*, $0.01 \cdot logn$) is always smaller than 1, *i.e.*, ERDC-SR is less expensive than RDC-SR in Setup. A second example is, we choose $c = 4600$ (*e.g.*, in PDP [8], if the malicious server corrupts 0.1% of the whole replica, the client needs to randomly check 4600 blocks in order to obtain the 99% data possession guarantee), $\alpha = 0.9$, $\tau = 12s$, $e = 5325\mu s$, and r_3 becomes $0.1 \cdot logn$. When n is larger than 2^{10}, r_3 will be larger than 1, *i.e.*, for this new set of c, α, τ, e parameters, ERDC-SR is more expensive than RDC-SR in Setup when the file has more than 2^{10} blocks.
- When $\beta < n$, r_3 is $\frac{t \cdot \frac{n}{2} \cdot log(\frac{2 \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha) \cdot e} + 1) \cdot e}{\frac{t \cdot n \cdot \tau}{c \cdot (1-\alpha)}}$, which can be further reduced to $\frac{log(\frac{2 \cdot ((1+\rho)^{\phi}-1) \cdot \tau}{c \cdot (1-\alpha) \cdot e} + 1) \cdot e \cdot c \cdot (1-\alpha)}{2 \cdot \tau}$. For example, we choose $c = 460$, $\alpha = 0.9$, $\tau = 12s$, $e = 5325\mu s$, and r_3 becomes $0.01 \cdot log(98 \cdot ((1+\rho)^{\phi} - 1) + 1)$. We show some concrete values of r_3 by varying ϕ and ρ in Table 5.7, from which we observe that for this set of c, α, τ, e parameters, r_3 is always smaller than 1, *i.e.*, ERDC-SR is less expensive than RDC-SR in Setup. A second example is, we choose $c = 4600$, $\alpha = 0.8$, $\tau = 12s$, $e = 5325\mu s$, and r_3 becomes $0.2 \cdot log(4.9 \cdot ((1+\rho)^{\phi} - 1) + 1)$. We also show some concrete values of r_3 by varying ϕ and ρ in Table 5.8, from which we observe that for this new set of c, α, τ, e parameters, ERDC-SR is more expensive than RDC-SR in Setup when ϕ is larger than or equal to 10 years.

We draw the following conclusion from the comparison of ERDC-SR and RDC-SR: Firstly, if the client targets a low data possession guarantee (*e.g.*, $c = 460$), ERDC-SR will be always less expensive than RDC-SR. For this case, we should always choose ERDC-SR. Secondly, if the client targets a high data possession guarantee (*e.g.*, $c = 4600$), we can differentiate two cases: 1) For the case of $\beta = n$, if the total number of PDP blocks in a file

is larger than a certain value (*e.g.*, 2^{10} in our example), ERDC-SR will be more expensive in both the Setup and the Repair phase. Thus, we should choose RDC-SR. Otherwise, we should choose ERDC-SR. 2) For the case of $\beta < n$, if the period of time ϕ (recall ϕ is a time period during which the client can obtain an assurance of the reliability and fault tolerance of the outsourced data) is larger than a certain value (*e.g.*, 10 years in our example), ERDC-SR will be more expensive in both the Setup and the Repair phase. Thus, we should choose RDC-SR. Otherwise, we should choose ERDC-SR.

Table 5.7 Concrete Values for r_3 by Varying ϕ and ρ When $c = 460$, $\alpha = 0.9$, $\tau = 12s$, $e = 5325\mu s$ (Recall That r_3 Is The Ratio between The Overall Computational Time of ERDC-SR and That of RDC-SR

ϕ \ ρ	30%	40%
5	0.08	0.09
10	0.10	0.11
15	0.12	0.14

Table 5.8 Concrete Values for r_3 by Varying ϕ and ρ When $c = 4600$, $\alpha = 0.8$, $\tau = 12s$, $e = 5325\mu s$ (Recall That r_3 Is The Ratio between The Overall Computational Time of ERDC-SR and That of RDC-SR

ϕ \ ρ	30%	40%
5	0.77	0.90
10	1.2	1.42
15	1.59	1.91

CHAPTER 6

AUDITABLE VERSION CONTROL SYSTEMS

This chapter introduces RDC-AVCS, an auditable version control system which relies on RDC to ensure that all the versions of a file are retrievable from the untrusted server over time. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, RDC-AVCS is the first to take a pragmatic approach for auditing real-world version control systems.

6.1 Introduction

Version control (also known as *revision control*) is the management of changes to collections of information, such as documents, computer programs, web pages, or configuration files. Version control provides the ability to track and control the changes made to the data over time. This includes the ability to recover an old version of a document. Software development often relies on a *Version Control System* (VCS) to automate the management of source code, documentation and configuration files. A VCS provides several useful features to software developers, such as: retrieve previous versions of the source code in order to locate and fix bugs, roll back to earlier versions in case the working version becomes corrupted, or allow team development in which multiple developers can work simultaneously on updates. In fact, a VCS is indispensable for managing large software projects. Popular version control systems include CVS [84], Subversion [85], Git [86], and Mercurial [87].

A version control system automates the process of version control. A VCS records all changes to the data into a data store called *repository*, so that any version of the data can be retrieved at any time in the future. Oftentimes, repositories are hosted by a third party, since they are potentially massive in size and cannot be stored and managed locally. For example, both Sourceforge [88] and Google Code [89] host repositories (based on Subversion or Git) for open-source projects, and GitHub [90] provides a paid service for Git repositories. Unfortunately, a third party is not necessarily trusted, for several reasons. Firstly, the service providers may rely on a public cloud storage platform, rather than an internal infrastructure, to host their users' data. For example, file hosting service providers like Dropbox [91], Bitcasa [92], that offer version control functionality to the stored data, use Amazon S3 [5] as a back-end storage service. Secondly, the service providers are vulnerable to various outside or even inside attacks. Thirdly, the service providers usually rely on complex distributed systems, which are vulnerable to various failures caused by hardware, software, or even administrative faults [93]. Additionally, unexpected accidental events may lead to the failure of services, *e.g.*, power outage [94, 95]. In Section 6.4.2, we provide additional arguments to support this threat model and the need to audit VCS systems.

Remote Data Checking (RDC) [7–9] can be used to address these concerns about the untrusted nature of a third party that hosts the VCS repository. RDC is a mechanism that has been recently proposed to check the integrity of data stored at untrusted third party providers of storage services. Briefly, RDC allows a client who initially stores a file with a storage provider to later check if the storage provider continues to store the original file in its entirety. This check can be done periodically, depending on the client's needs.

From the data owner's point of view, it should be possible to retrieve any previous version of the data, even if the repository is hosted at an untrusted VCS server. In a straightforward application of RDC, if a file `F` has t versions, $\texttt{F}_0$ through $\texttt{F}_{t-1}$, then each file version can be seen as an independent file and the client can use RDC independently to check the integrity of each file version. This solution, unfortunately, has prohibitive costs for several reasons. VCS repositories may store many versions and storage overhead would be very large if every version is stored in its entirety (*e.g.*, the source code for the `gcc` compiler [96] has over 200,000 versions). Moreover, the RDC costs associated with creating metadata and checking each version independently would be too large.

To reduce the storage overhead, modern version control systems adopt "delta encoding" to store versions in a repository: Only the first version of a file is stored in its entirety, and each subsequent version of the file is stored as the difference from the immediate previous version. These differences are recorded in discrete files called "deltas". Thus, if there are t versions of a file, the VCS server stores them as the initial file and $t-1$ deltas. A popular version control system that uses a variant of delta encoding is Git [86]. Delta encoding optimizes the storage required to represent all the versions of a file. Such a delta encoded repository is not optimized towards retrieving individual versions: To retrieve version t, the VCS server starts from the initial version and applies all subsequent deltas up to version t, thus incurring a cost linear in t. Considering that source code repositories may have hundreds of thousands of versions (*e.g.*, `GCC` [96]), retrieving an arbitrary version can be burdensome on the server.

Skip delta encoding is a type of delta encoding which is further optimized towards reducing the cost of retrieval. A new file version is still stored as the difference from a previous file version. This difference is not relative to the immediate previous version, but

it is relative to another previous version (more details in Section 6.3.1). This ensures that retrieval of the t-th version only requires $log(t)$ applications of deltas by the VCS server. A popular VCS that uses skip delta encoding is Apache Subversion (in short, SVN) [85].

The evolution of a file managed with a VCS can be seen as a sequence of updates, each update resulting in a new file version. As such, the integrity of a VCS repository could be verified using an RDC protocol designed to allow dynamic updates to the data. Several RDC schemes can handle the full range of dynamic update operations [12, 13], such as modifications, insertions, and deletions. A dynamic RDC scheme can directly be used to check the integrity of the latest file version (every new file version can be seen as a series of updates to the previous file version). A dynamic RDC scheme can also be adapted to check the integrity of the entire VCS repository – basically check all versions of a file – by organizing the file versions in an authentication structure.

Using a dynamic RDC scheme to check the integrity of a VCS repository has several important drawbacks:

– *First*, all the real-world VCS systems *require only the append operation* – the repository stores the initial file version and a series of deltas for subsequent versions, all of which can be seen as append operations to the initial version. Thus, using a full-fledged dynamic RDC scheme that supports the full range of updates is overkill and incurs additional unnecessary overhead during the Challenge and Commit phases, as illustrated in Table 6.1. DPDP and DR-DPDP are built on top of delta-based version control systems, whereas the designed RDC-AVCS scheme is built on top of skip delta-based version control systems. Indeed, previous work on checking integrity of version control systems [12, 65, 68] extends a dynamic RDC scheme which relies on a tree-like structure, thus adding a logarithmic cost to the Challenge and Commit phases. However, the only meaningful operation for modern VCS systems (*e.g.*, CVS, SVN, Git) is the append operation, since they are designed to keep a record of all the data in all previous versions.
– *Second*, a dynamic RDC scheme that supports the full range of dynamic updates has a higher complexity than an RDC scheme designed to only support appends at the end of the file. The additional complexity brings with it a more complex adversarial model and a more complex proof of security, all of which make the scheme more prone to security and implementation flaws.

Table 6.1 Comparison of Different RDC Schemes for Version Control Systems

	DPDP [12]	DR-DPDP [68]	RDC-AVCS
Communication (Commit phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Server computation (Commit phase)	$O(n + log(t))$	$O(n)$	$O(nlog(t))$
Client computation (Commit phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Communication (Challenge phase)	$O(logn + log(t))$	$O(1 + logn)$	$O(1)$
Communication (Retrieve phase)	$O(n + log(t))$	$O(n + 1)$	$O(n + 1)$
Server computation (Retrieve phase)	$O(tn + log(t))$	$O(tn + 1)$	$O(nlog(t) + 1)$
Client computation (Retrieve phase)	$O(n + log(t))$	$O(n)$	$O(n)$
Client storage	$O(n)$	$O(n)$	$O(n)$
Server storage	$O(nt)$	$O(nt)$	$O(nt)$

6.2 Related Work

Remote data checking for archival storage. As an effective technique for ensuring the integrity of data outsourced at an untrusted party, remote data checking (RDC) has been investigated extensively for both the single-server setting ([8–11, 97, 98]) and the multiple-server setting ([21–23, 28]). Recent work on RDC focuses on new topics such as proofs of fault tolerance [69], proofs of location [63, 70, 71] and server-side repair [26].

Dynamic remote data checking. Dynamic Provable Data Possession (DPDP) relies on authenticated data structures (e.g., skip lists [12], RSA trees [12], Merkle trees [13], 2-3 trees [14]) to support the full range of dynamic operations. DPDP adopts spot checking for efficiency and is thus vulnerable to small corruption attack. Follow-up work [24,25] tries to mitigate such an attack by adding robustness. Concurrently with DPDP, Dynamic Proofs of Retrievability (D-PoR) tries to adapt PoR to a dynamic setting. To support D-PoR, recent work either computes and stores the parity of the data at the client side [15], or relies on Oblivious RAM [36].

Remote data checking for version control systems. Anagnostopoulos et al. [99] introduced the notion of persistent authenticated dictionaries, which allow the user to

check whether element e was on set S at time t. Erway et al. [12] adopted a two-level authenticated data structure to provide integrity guarantee for version control systems. Specifically, for each file version, a first-level authenticated data structure is used to organize all of its blocks, generating a root for each version. A second-level authenticated data structure is then used to organize all of these roots. The checking complexity is thus $O(log(tn))$, in which t is the total number of versions and n is the total number of blocks in a version. Etemad et al. [68] improved the solution proposed in [12]. They adopt a PDP-like structure [8], rather than an authenticated data structure, to provide integrity guarantee for the roots of the first-level authenticated data structure, thus reducing the checking complexity to $O(1 + log(n))$. Zhang et al. [65] proposed an update tree-based solution. Their scheme adopts a tree structure to organize all the update operations, and thus the checking complexity is logarithmic in the total number of updates, *i.e.*, approximately $O(log(t))$. In RDC-AVCS, we provide the most efficient solution known to date, which relies solely on an efficient RDC scheme to reduce the checking complexity to $O(1)$.

6.3 Background on Version Control Systems and Remote Data Checking

6.3.1 Version Control Systems

Software development relies on a *Version Control System (VCS)* to automate the management of source code, documentation and configuration files. Typically, one (or more) VCS clients interact with a VCS server and the VCS server stores all the changes to the data into a *main repository*, such that any prior version of the data can be retrieved at any time in the future. Each VCS client has a local repository, which stores the *working copy*, the changes made by the client to the working copy, and some metadata. The working

copy is the version of the data that was last checked out by the client from the main VCS repository.

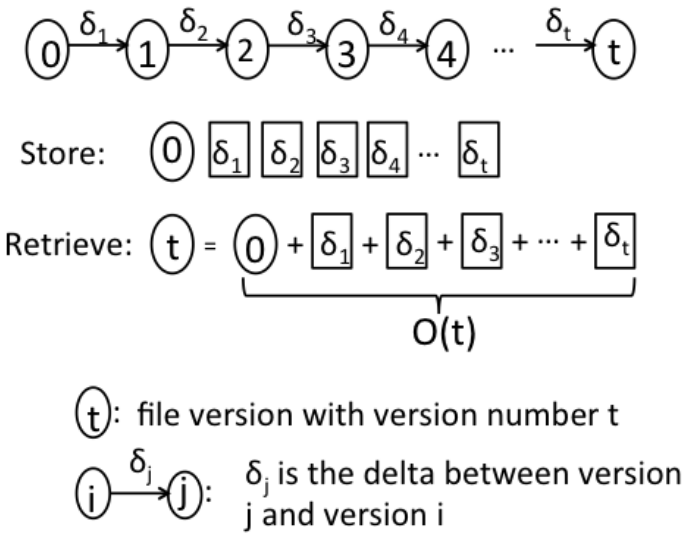
A VCS provides several useful features to track and control the revisions (changes) made to the data over time. This includes operations such as commit, update, revert, branch, merge, and log. In practice, the most commonly used operations by a VCS client are *commit* and *retrieve*. Commit refers to the process of submitting the latest changes of the data to the main repository, so that the changes to the working copy become permanent. Retrieve refers to the process of replacing the working copy with an older or a newer version stored on the server.

Delta-based VCS. With a version control system, the data owner would like to keep every change of her data in the repository, so that at any point of time in the future, she can revert to a previous version, or update to a new version. One simple solution is to store a new version of the data in its entirety upon each commit (*e.g.*, CVS [84] adopts this method for binary files). Such a straightforward solution has large communication and storage overhead, since in most cases, only a small portion of the whole data has been updated; thus, sending and storing the whole new version may result in significant unnecessary communication and storage.

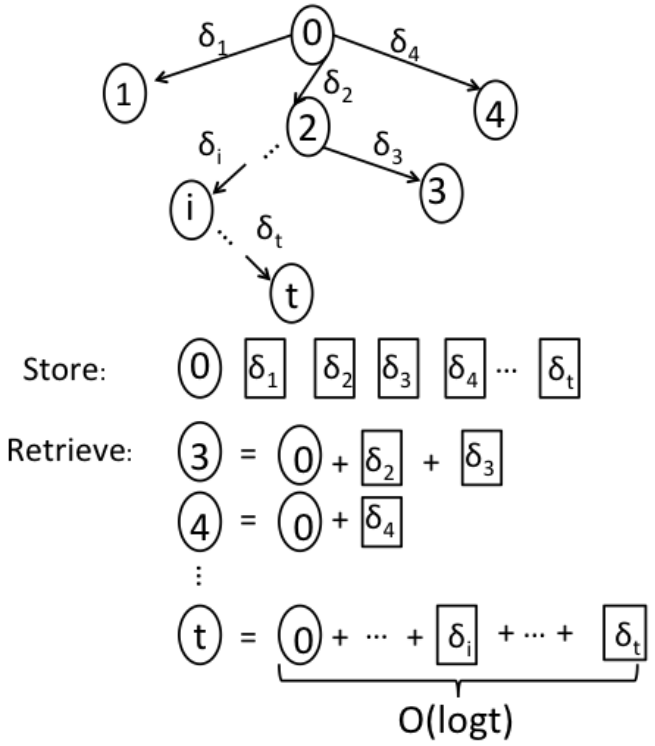
To reduce the storage overhead, modern VCS systems adopt "*delta encoding*" to store changes to the data in the repository: Only the first version of a file is stored in its entirety, and each subsequent version of the file is sent and stored as the difference from the immediate previous version. These differences are recorded in discrete files called "deltas". Thus, if there are t versions of a file, the VCS server stores them as the initial file and $t - 1$ deltas (see Figure 6.1(a)). Popular version control systems that use variants

of delta encoding are Git [86], SVN [85] and CVS [84][1]. Delta encoding optimizes the storage required to represent all the versions of a file, but a delta encoded repository is not optimized towards retrieving individual versions: To retrieve version t, the VCS server starts from the initial version and applies all subsequent deltas up to version t, thus incurring a cost linear in t (again, see Figure 6.1(a)). Considering that source code repositories may have hundreds of thousands of versions (*e.g.*, `GCC` [96]), retrieving an arbitrary version can be burdensome on the server.

[1]CVS uses delta encoding only for text files



(a) Delta-based VCS



(b) Skip delta-based VCS

Figure 6.1 Delta-based and skip delta-based version control systems.

Skip delta-based VCS. Skip delta encoding is a type of delta encoding which is further optimized towards reducing the cost of retrieval. A new file version is still stored as

the difference from a previous file version; however, this difference is not relative to the immediate previous version, but it is relative to a certain previous version. This ensures that retrieval of the t-th version only requires $log(t)$ applications of deltas by the VCS server. A popular VCS that uses skip delta encoding is Apache Subversion (in short, SVN) [85].

In this case, the difference is called a "skip delta" and the old version against which a new version is encoded is called a "skip version". When version i is committed, the skip delta is computed against the skip version j. The rule for selecting the skip version j is: Consider the binary representation of i and change the rightmost bit that has value "1" into a bit with value "0". For example, in Figure 6.1(b), version 4's skip version is version 0, because the binary representation of 4 is 100, and by changing the rightmost "1" bit into a "0" bit, we get 0.

By adopting the skip delta-based approach, the cost to recover any version is logarithmic in the total number of versions. For example, in Figure 6.1(b), to reconstruct version 3, start from version 0 and apply δ_2 and δ_3; to reconstruct version 4, start from version 0 and apply δ_4. The skip version for version 25 is 24, whose skip version is 16, whose skip version is 0. Thus, to reconstruct version 25, start from version 0 and apply $\delta_{16}, \delta_{24}, \delta_{25}$. In Appendix E.1, we show that the cost for retrieving an arbitrary version t is bounded by $O(log(t))$.

6.3.2 Remote Data Checking

Remote Data Checking (RDC) allows the data owner to check the integrity of data outsourced at an untrusted server, and thus to audit whether the server fulfills its contractual obligations. A remote data checking protocol consists of three phases: Setup, Challenge, and Retrieve. Consider that the storage of one file is outsourced at an untrusted server.

Then, during the Setup phase, the data owner preprocesses the file $\mathtt{F}$ and generates verification metadata Σ, and then stores both $\mathtt{F}$ and Σ at the untrusted server. The data owner then deletes $\mathtt{F}$ and Σ from its local storage and only keeps a small constant amount of secret key material K. During the Challenge phase, a verifier (the data owner or a third-party verifier) challenges the server to prove that it really possesses the file previously stored by the data owner. The server generates a proof of possession based on the stored file and metadata, and sends back the proof. The client then checks the proof based on the key material K. During Retrieve, the data owner recovers the original file.

PDP (Provable Data Possession [8]) and PoR (Proofs of Retrievability [9, 10]) are two examples of RDC protocols. In PDP/PoR, during the Setup phase, the data is seen as a collection of fixed-size blocks, and the client computes a tag for each block. During the Challenge phase, the verifier randomly checks the integrity of a random subset of the file blocks. The Challenge phase can be very efficient: For example, it is shown [8] that if the server corrupts a certain fraction of the file (*e.g.*, 1%), the verifier can detect such corruptions with high probability by only randomly checking a constant number of blocks; in this case, the communication between the verifier and the server is also constant in size.

PDP/PoR have been shown to be extremely efficient during the Challenge phase [8–10], with constant communication and constant client/server computation. Both PDP and PoR have been originally proposed for archival storage and only support static data. Later, a more complex PDP protocol was proposed to support dynamic operations on the outsourced data, such as insertions, deletions and modifications [12]. In Section 6.6.1 we show that RDC schemes for static data can securely support one specific dynamic operation, namely append at the end of the file. In Section 6.5, we build an RDC scheme for skip

delta-based VCS systems, which relies on any RDC scheme that supports block appends at the end of the file.

6.4 Model and Guarantees

6.4.1 System Model

An Auditable Version Control System (AVCS) is a version control system (VCS) designed to function under an adversarial setting. In AVCS, just like in a regular VCS, one or more clients store data at a server. The server maintains the *main repository*, where all the versions of the data are stored. Each client runs an AVCS client software. In this work, we use the term client to refer to the AVCS client software and server to refer to the AVCS server software. Each AVCS client has a local repository, which stores the working copy, the changes made by the client to the working copy, and some metadata. The working copy is the version of the data that was last checked out by the client from the main VCS repository.

From a client's point of view, the interface exposed by the server includes two main operations: *commit* and *retrieve*[2]. Commit refers to the process of submitting the latest changes of the data to the main repository, so that the changes in the client's working copy become permanent. Retrieve refers to the process of replacing the client's working copy with an older or a newer version stored on the server.

AVCS incorporates all the functionality offered by a regular VCS. In addition, the AVCS server exposes one additional operation, *check*, which permits the client to check if the server possesses all the versions of a file.

[2]VCS systems permit additional operations such as *branch*, *merge*, *log*, etc., but in this work we focus on *commit* and *retrieve*, which are the most common operations.

The AVCS main repository may contain several projects. Each project may contain one or more files. For each file, the changes submitted by the client are stored by the server using delta encoding, as described in Section 6.3.1. Each change is stored as a discrete "delta" file. So, if there are $t-1$ changes for a file, then the server will store the initial version of the file and $t-1$ delta files, $\delta_1, ..., \delta_{t-1}$. We focus our discussion on storing, checking, and retrieving the versions of one file; this can be easily generalized to multiple files.

6.4.2 Adversarial Model

We consider a threat model in which there are no malicious clients, *i.e.*, all clients are trusted. Meanwhile, the server is not trusted and may misbehave [8]. This captures a setting in which the employees of a company collaborate on a software development project (so they are all trusted), but the AVCS server is outsourced at a third party which is not necessarily trusted. The server may misbehave as follows:

- It may reclaim storage by discarding data that is rarely accessed (economically motivated), or try to hide data loss incidents to preserve its reputation. Data loss incidents may be accidental (e.g., administrative errors, hardware and software failures) or malicious (*e.g.*, insider or outsider attacks).
- During retrieve, it may not provide the requested version correctly, *e.g.*, it may provide a corrupted version, or a version which is either older or newer than the requested version. Possible reasons for such misbehavior could be: The repository has been corrupted (accidentally or maliciously), or the server has reclaimed some rarely accessed data, or the server-side software does not function properly, etc.

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

gain. In particular, we do not consider attacks in which the server simply corrupts a small portion of the repository (*e.g.*, 1 byte), because saving such a small amount of storage will not provide a significant benefit for the server. For a discussion about protection against small corruption attacks, see Section 6.6.

The server is assumed to at least respond to the client's requests. Otherwise, if the server is non-responsive, the client will terminate its contract with the server and choose another service provider. To protect the client-server communication against external adversaries, we assume that this communication occurs over secure channels, *e.g.*, the communication is secured using SSL/TLS.

On the importance of auditing VCS systems. We provide several arguments to motivate this threat model and to highlight the importance of auditing VCS systems:

- Even though source code repositories are not very large (*e.g.*, the entire `gcc` repository is about 1GB), popular hosting services have a huge number of repositories. In 2013, GitHub hosted over 6 million repositories [100], SourceForge over 324,000 projects [101] and Google Code over 250,000 projects. It is conceivable that some service providers may be economically motivated to misbehave.
- The techniques we propose are applicable to all VCS-es that rely on skip delta encoding, including those that store other type of data than source code. For example, Dropbox saves the history of all deleted and earlier versions of files (free for 30 days, and unlimited deletion recovery and version history with the "Packrat" option).
- There are ongoing efforts to add support for large media binary files into VCS-es like Git [102, 103].
- Hosting providers like Dropbox [91] and Bitcasa [92] that offer version control functionality rely on cloud storage services like Amazon S3 as the back-end storage. It is conceivable that even providers like GitHub may adopt a similar model in the future. There is plenty of evidence that cloud service providers should not be fully trusted.

6.4.3 Security Guarantees

Consider an AVCS repository which contains t versions of the file $\mathtt{F}$ (these are stored in the repository as the initial version of the file $\mathtt{F}_0$ and $t-1$ delta files, δ_1, ..., δ_{t-1}). Let $\tilde{\mathtt{F}}$ be the virtual file obtained by concatenating $\mathtt{F}_0$, δ_1, ..., δ_{t-1}, *i.e.* $\tilde{\mathtt{F}} = \mathtt{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. We seek to build AVCS systems which provide the following security guarantees:

- **SG1** (Data Possession): Upon checking the integrity of all the versions of $\mathtt{F}$ stored in the repository, the client can detect if the server corrupts a fraction of $\tilde{\mathtt{F}}$.
- **SG2** (Version Correctness): Upon retrieving $\mathtt{F}_i$ (version i of $\mathtt{F}$) from the server, the client can verify the correctness of $\mathtt{F}_i$, for any $i \in [0, t-1]$.

The practical implications of these guarantees are that the server cannot corrupt some of the file's versions without being detected and that it cannot serve an incorrect file version to the client. **SG1** captures the client's ability to check if the server continues to possess all of the versions of $\mathtt{F}$ that have been stored in the main repository. **SG2** captures the client's ability to detect if the server provides a corrupt version, or a version that is different than the version requested by the client.

6.5 Auditable Version Control Systems (AVCS)

In this section, we first give an overview of VCS systems designed to work under a benign setting. We then introduce the definition of *Auditable Version Control Systems* (AVCS), which are VCS systems designed to function under an adversarial setting, and propose a construction based on remote data checking mechanisms.

Notation. The VCS repository contains t versions of the file $\mathtt{F}$, which are stored in the repository as $\mathtt{F}_0, \delta_1, \delta_2, ..., \delta_{t-1}$. $\mathtt{F}_0$ is the initial version of the file, and the $t-1$ delta files are based on skip delta encoding as described in Section 6.3.1. We focus our discussion on

storing, checking, and retrieving the versions of one file; this can be generalized to multiple files.

We use $\mathtt{F}_i$ to denote version i of the file. We use $\mathtt{F}_{skip(t)}$ to denote the skip version for $\mathtt{F}_t$ (the algorithm for determining $\mathtt{F}_{skip(t)}$ is described in Section 6.3.1). We write $\mathtt{F}_i = \mathtt{F}_j + \delta$ to denote that $\mathtt{F}_i$ is obtained by applying δ to $\mathtt{F}_j$.

6.5.1 Skip Delta-based Version Control Systems

Version control systems which use skip delta encoding have been designed for a benign setting, in which the VCS server is assumed to be fully trusted. A popular VCS which relies on skip delta encoding is Apache Subversion [85] (in short, SVN), described on its website as an "open-source, centralized version control system characterized by its reliability as a safe haven for valuable data".

The main operations of such VCS systems fall under three phases: Setup, Commit, and Retrieve, as follows:

In the Setup phase, the client (data owner) contacts the server to create a new project in the main VCS repository[3]. For example, in SVN, this can be achieved using the command "`svn import`", which will create a new project in the main VCS repository using a codebase that exists at the client – this will be the first version of the project. The client will then create its local working copy by checking out this first version from the server, using the command "`svn checkout`".

In the Commit phase, the client commits the changes in its local working copy into the main VCS repository. For example, in SVN, this can be achieved using the command "`svn commit`". The client wants to commit a new version, $\mathtt{F}_t$ (note that the client also

[3]We assume that an (empty) VCS repository has been already created, *e.g.*, by using the SVN command "`svnadmin create`".

has a local copy of $\mathtt{F}_{t-1}$, which is the working copy). Then the client computes the "delta" between $\mathtt{F}_t$ and $\mathtt{F}_{t-1}$, *i.e.* δ such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$, and sends δ to the server. After receiving δ, the server executes:

1. Compute $\mathtt{F}_{t-1}$ based on data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply skip deltas $\ldots, \delta_i, \ldots, \delta_{t-1}$).
2. Compute $\mathtt{F}_t$ based on $\mathtt{F}_{t-1}$ and δ: $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$.
3. Compute the skip version $\mathtt{F}_{skip}$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply skip deltas $\ldots, \delta_i, \ldots, \delta_{skip}$).
4. Compute δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{skip} + \delta_{skip}$, and store δ_{skip} as δ_t in the repository.

In the Retrieve phase, the client retrieves an arbitrary version of the data. For example, in SVN, this can be achieved using the "`svn update -r` i" command. The client wants to replace version j (the working copy) with version i. The server executes:

1. Compute $\mathtt{F}_i$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply the corresponding skip deltas).
2. Compute $\mathtt{F}_j$ based on the data in the repository (*i.e.*, start from $\mathtt{F}_0$ and apply the corresponding skip deltas).
3. Compute δ such that $\mathtt{F}_i = \mathtt{F}_j + \delta$.
4. Return δ to the client.

The client then computes $\mathtt{F}_i$: $\mathtt{F}_i = \mathtt{F}_j + \delta$.

6.5.2 Definition of An AVCS System

The previous section described the behavior of a skip delta-based VCS system in a benign setting, where the VCS server is fully trusted and does not deviate from the protocol. In this work, we consider a setting in which the VCS server is untrusted (the adversarial model is

described in Section 6.4). We propose an *Auditable Version Control System* (AVCS), which is a delta-based VCS enhanced to work in an adversarial setting.

An AVCS scheme consists of seven polynomial-time algorithms (KeyGen, ComputeDelta, GenMetadata, GenProof, CheckProof, GenRetrieveVersionAndProof, CheckRetrieveProof). KeyGen is a key generation algorithm run by the client to setup the scheme. ComputeDelta is run by the client to compute a delta when committing a new file version. GenMetadata is run by the client to generate the verification metadata for a new file version, before committing the new version. GenProof is run by the server and CheckProof is run by the client in order to generate and verify a proof of data possession, respectively. Similarly, GenRetrieveVersionAndProof is run by the server and CheckRetrieveProof is run by the client to retrieve an arbitrary file version.

An AVCS system has four phases: Setup, Commit, Challenge, and Retrieve.

- Setup: The client runs KeyGen to generate the private key material and performs other initialization operations.
- Commit: To commit a new file version, the client runs ComputeDelta and GenMetadata to compute the delta and the metadata for the new file version, respectively. The delta and the metadata are both sent to the server.
- Challenge: Periodically, the verifier (client) challenges the server to obtain a proof that the server continues to store all the file versions committed by the client. The server uses GenProof to compute a proof of data possession, and the client uses CheckProof to validate the proof.
- Retrieve: The client requests an arbitrary version of the stored data. The server runs GenRetrieveVersionAndProof to obtain the requested file version, together with a proof of correctness. The client verifies the correctness of the file retrieved from the server by running CheckRetrieveProof.

Note that this definition encompasses VCS systems that use delta encoding. This includes skip delta-based VCS systems.

6.5.3 RDC-AVCS: An Auditable Version Control System based on Remote Data Checking

In this section, we present our main result, RDC-AVCS, the first auditable version control system. RDC-AVCS is obtained by integrating RDC mechanisms into a VCS system. Whereas our definition of AVCS targets VCS systems that use delta encoding in general, in our RDC-AVCS construction we focus on VCS systems that use skip delta-based encoding. As explained in Section 6.3.1, these are optimized for both storage and retrieval; however, they are arguably more challenging to secure than VCS systems that use delta encoding, because of the nature of computing the skip deltas.

Challenges. Going from a benign setting to an adversarial setting, we need to overcome several challenges. These challenges stem from the adversarial nature of the VCS server and from the format of a skip delta-based VCS repository which is optimized to minimize the server's storage and workload during the Retrieve phase:

– *The gap between the server's and the client's view of the repository.* In a general-purpose RDC protocol (Section 6.3.2), the client and the server have the same view of the outsourced data: the client computes the verification metadata based on the data, and then sends both data and metadata to the server. The server stores these unmodified. The server then uses the data and metadata to answer the client's challenges by computing a proof that convinces the client that the server continues to store the same data outsourced by the client.

In a skip delta-based VCS, there is a gap between the two views, which makes skip delta-based VCS systems more difficult to audit: Although both client and server view the main VCS repository as the initial version of the data plus a series of delta files corresponding to subsequent data versions, they have a different understanding of the delta files. To commit a new version t, the client computes and sends to the server a delta that is the difference between the new version and its immediate previous version, that is the difference between version t and $t-1$ (recall that the client only stores the working copy which is version $t-1$, and version t which incorporates the changes made by the client over version $t-1$). This is different from the skip deltas that are stored by the server: a δ_i file stored by the server is the difference between version i and a "skip version", which is not necessarily the immediate version previous to i. For example, the skip delta for version 128 will be computed as the difference against version 0 (the algorithm for selecting the

"skip version" is described in Section 6.3.1). Since the client does not have access to the skip deltas stored by the server, it cannot compute the verification metadata over them, as needed in an RDC protocol.
– *Delta encoding is not reversible.* The client may try to retrieve the skip delta computed by the server and then compute the verification metadata based on the retrieved skip delta. Unfortunately, in an adversarial setting, the client cannot trust the server to provide a correct skip delta value. This is exacerbated by the fact that delta encoding is not a reversible operation. If $\delta_{t-1\to t}$ is the difference between versions $t-1$ and t (*i.e.*, $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta_{t-1\to t}$), this does not imply that $\mathtt{F}_{t-1}$ can be obtained based on $\mathtt{F}_t$ and $\delta_{t-1\to t}$. The reason comes from the method used by delta encoding to encode update operations between versions, such as insert, update, delete. If a delete operation was executed on version $t-1$ to obtain version t, then $\delta_{t-1\to t}$ encodes only the position of the deleted portion from $\mathtt{F}_{t-1}$, so that given $\mathtt{F}_{t-1}$ and $\delta_{t-1\to t}$, one can obtain $\mathtt{F}_t$. Since $\delta_{t-1\to t}$ does not encode the actual data that has been deleted, $\mathtt{F}_{t-1}$ cannot be obtained based on $\mathtt{F}_t$ and $\delta_{t-1\to t}$.

A first attempt. We make two observations which we then leverage to build an initial, alas inefficient AVCS system:

– *First*, we observe that any RDC protocol that supports the append operation securely can be used to audit the integrity of a VCS server that relies on skip delta encoding, simply because RDC can be used to spot check the blocks of a virtual file obtained by concatenating the original file and the subsequent delta files. In Section 6.6.1, we show that existing RDC protocols proposed for static data can be enhanced to securely support the append operation.
– *Second*, we need to unify the client's and server's views of the repository data so that the client can compute on its own the metadata over the delta files that are stored at the server.

To bridge the gap between the server's and the client's view of the repository, we require that, upon each commit, the skip delta is computed by the client and not by the server. The client will then send the skip delta to the server, together with RDC verification tags computed over the skip delta. To be able to compute the skip delta, the client should store several previous versions, so that it has access to the "skip version" against which the skip delta is computed. Theorem 6.5.1 shows that, unfortunately, the storage required for storing enough previous versions on the client side is linear with the total number of versions in a repository. This does not conform with our notion of outsourcing the VCS repository, in which the client should only store one version of the file (the working copy).

Theorem 6.5.1. *The client storage for the inefficient* AVCS *system is* $O(t)$*, in which* t *is the total number of versions in a repository.*

Proof. (sketch) Let $f(t)$ be the total number of versions needed to be stored in the client to facilitate the computation of skip deltas in the inefficient AVCS system. Let $i \leftarrow j$ denote that version i is version j's skip version; similarly $i \rightarrow j$ denotes version j is version i's skip version. Let $b_0 \ldots b_i \ldots b_{t-1}$ be the binary representation of a version number t, in which b_i is either "0" or "1", *e.g.*, 00 is version number 0's binary representation.

- For $t = 4$, according to the rule of determining the skip version, we have: $01 \rightarrow 00 \leftarrow 10 \leftarrow 11$. One can observe that by only storing version 0, the client can always compute all the skip deltas locally: The client can compute locally the skip deltas for versions 1 and 2, since the skip version for both of these is version 0; The client can also compute locally the skip delta for version 3, since version 2 (which is the skip version for version 3), is version 3's immediate previous version. In other words, $f(4) = 1 = 2^0 = 2^{log4-2}$.
- For $t = 8$, one can simply divide all the 8 versions into 2 groups: Group 1, in which the first bit is 0: 001 $\rightarrow$ 000 $\leftarrow$ 010 $\leftarrow$ 011; Group 2, in which the first bit is 1: 101 $\rightarrow$ 100 $\leftarrow$ 110 $\leftarrow$ 111. Without considering the first bit, each of the aforementioned two groups is equivalent to the case of $t = 4$, thus, $f(8)$ should be twice compared to $f(4)$: $f(8) = 2 * f(4) = 2 = 2^1 = 2^{log8-2}$. Similarly, for the general case, we have: $f(t) = 2 * (f(\frac{t}{2}))$, by which we can further compute that $f(t) = 2^{log(t)-2} = \frac{t}{4}$.

□

The RDC-AVCS **Construction** We are now ready to present RDC-AVCS, an auditable VCS scheme which uses RDC mechanisms to ensure all the versions of a file can be retrieved from the VCS server. RDC-AVCS only requires the same amount of storage on the client like a regular VCS system. This scheme is the main result of this work.

Recall that the VCS repository contains t versions of the file, $\mathtt{F}_0, \mathtt{F}_1, ..., \mathtt{F}_{t-1}$. The t versions are stored in the repository as t files: $\mathtt{F}_0$, δ_1, δ_2, ..., δ_{t-1} (*i.e.*, the initial version of the file and $t - 1$ skip delta files).

For the purpose of our scheme, we view all the information pertaining to the versions of the file $\mathtt{F}$ as a virtual file $\tilde{\mathtt{F}}$ obtained by concatenating the original file and the subsequent delta files: $\tilde{\mathtt{F}} = \mathtt{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. We view $\tilde{\mathtt{F}}$ as a collection of fixed-size blocks, each block containing s symbols, and each symbol is an element of $GF(p)$, where p is a large prime (at least 80 bits). This view matches the view of a file in an RDC scheme: To check the integrity of all the versions of $\mathtt{F}$, it is enough to check the integrity of $\tilde{\mathtt{F}}$. Let n denote the number of blocks in $\tilde{\mathtt{F}}$. As the client commits new file versions, n will grow accordingly (note that n is maintained by the client).

RDC-AVCS **overview.** We use two types of verification tags. To check data possession (in the Challenge phase) we use *challenge tags*; these are computed over the blocks in $\tilde{\mathtt{F}}$ to facilitate spot checking in RDC [8]. To check the integrity of individual file versions (in both the Commit and the Retrieve phases), we use *retrieve tags*; these are computed over entire versions of $\mathtt{F}$.

To check the integrity of $\tilde{\mathtt{F}}$, we adopt the challenge tags introduced by Shacham and Waters [10][4]. When the client commits a new file version, it computes a retrieve tag in the form of a MAC over the whole file version that is to be committed. This retrieve tag will be stored at the VCS server and will be used by the server to convince the client of file version integrity during Commit and Retrieve.

In a benign setting, whenever the client commits a new file version, the server computes and stores a skip delta file in the main VCS repository (as described in Section 6.5.1). Under an adversarial setting, to leverage RDC techniques over the VCS repository, the skip delta files must be accompanied by verification challenge tags. Since

[4]For efficiency reasons, we use the tags that support private verifiability. Our scheme could also be instantiated using the challenge tags in [10] that are publicly verifiable.

the challenge tags can only be computed by the client, our scheme requires the client to obtain the skip delta, compute the challenge tags over it and send both the skip delta and the tags to the server.

When committing a new version $\mathtt{F}_t$, the client must compute the skip delta (δ_{skip}) for $\mathtt{F}_t$. The δ_{skip} must be computed against a certain previous version of the file, called the "skip version" (as described in Section 6.3.1). Recall that the client also has in its local store a copy of $\mathtt{F}_{t-1}$, the working copy.

If $(skip(t) == t - 1)$, then the client can directly compute δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta_{skip}$. Otherwise, the client computes δ_{skip} by interacting with the VCS server as follows:

1. The client computes the difference between the new version and the immediate previous version, *i.e.* computes δ such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$. The client sends δ to the server.
2. The server re-computes $\mathtt{F}_{t-1}$ based on the data in the repository and then computes $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$. The server then re-computes $\mathtt{F}_{skip(t)}$ (the skip version for $\mathtt{F}_t$) based on the data in the repository and computes the difference between $\mathtt{F}_t$ and $\mathtt{F}_{skip(t)}$, *i.e.* it computes $\delta_{reverse}$ such that $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$. The server sends $\delta_{reverse}$ to the client, together with the retrieve tag for $\mathtt{F}_{skip(t)}$.
3. The client computes the skip version: $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$ and checks the validity of $\mathtt{F}_{skip(t)}$ using the retrieve tag received from the server. The client then computes the skip delta for the new file version, *i.e.* δ_{skip} such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$.

To give an example, when the client commits $\mathtt{F}_{15}$, the client also has the working copy $\mathtt{F}_{14}$ which is the skip version for $\mathtt{F}_{15}$, and the client can compute directly δ_{skip} such that $\mathtt{F}_{15} = \mathtt{F}_{14} + \delta_{skip}$. However, when the client commits $\mathtt{F}_{20}$, it only has $\mathtt{F}_{19}$ in her local store and must first retrieve from the server $\delta_{reverse}$ and then compute $\mathtt{F}_{16}$ which is the skip version for $\mathtt{F}_{20}$, as $\mathtt{F}_{16} = \mathtt{F}_{20} + \delta_{reverse}$. Only then can the client compute δ_{skip} such that $\mathtt{F}_{20} = \mathtt{F}_{16} + \delta_{skip}$.

For the Challenge phase, we leverage a mechanism based on checking the integrity of the remotely stored data, like in previous RDC schemes [8, 9]. With every challenge,

the client challenges the server to prove possession of a random subset of the blocks in $\tilde{\mathrm{F}}$. The server provides a proof of possession which convinces the client that the server can produce the data in the challenged blocks. This spot checking mechanism is quite efficient. For example, when the server corrupts 1% of the repository (*i.e.*, 1% of $\tilde{\mathrm{F}}$), then the client can detect this corruption with high probability by randomly checking only a small constant number of blocks (*e.g.*, checking 460 blocks results in a 99% detection probability) [8].

In the Retrieve phase, the client replaces her working copy with another file version. The client can use the corresponding retrieve tag to check the correctness of the file version provided by the server.

The RDC-AVCS scheme. The details of the RDC-AVCS scheme are presented in Figures 6.2, 6.3 and 6.4. Let $\tilde{\mathrm{F}}$ be a virtual file obtained by concatenating the original file and the subsequent delta files: $\tilde{\mathrm{F}} = \mathrm{F}_0||\delta_1||\delta_2||...\delta_{t-1}$. Let n be the number of blocks in $\tilde{\mathrm{F}}$. The client maintains n and updates n accordingly whenever she commits a new file version to the repository.

The Setup phase. The client runs KeyGen to generate two private keys K_1 and K_2, and picks s random numbers from $GF(p)$, which will be used in computing the challenge tags. The client also sets $n = 0$.

The Commit phase. To commit a new file version, the client uses ComputeDelta to compute the skip delta for the new file version, and runs GenMetadata to generate the corresponding challenge and retrieve tags. In ComputeDelta (Figure 6.3), the client first uses SelectSkipVersion to determine the skip version. If the skip version is the immediate previous version of the new version, the client simply computes the skip delta based on the new version and its immediate previous version. Otherwise, the client contacts the server,

sending the delta of the new version against its immediate previous version. The server uses ComputeReverseAndSkipDelta to generate the delta of the skip version against the new version, *i.e.*, $\delta_{reverse}$, and returns to the client $\delta_{reverse}$ and the retrieve tag of the skip version. The client then re-computes the skip version based on the new version and $\delta_{reverse}$, and verifies the validity of the computed skip version by running CheckRetrieveProof. If the verification succeeds, the client computes the skip delta based on the new version and the skip version. After having computed the skip delta, the client runs GenMetadata (Figure 6.3) to compute the challenge tags and the retrieve tag, which will then be sent to the server. The retrieve tag $\mathtt{T}_t$ is computed using an HMAC function [104]. Finally, the client increases n by d, where d is the number of blocks in the skip delta.

The Challenge **phase.** Periodically, the client challenges the server to prove possession of the virtual file $\tilde{\mathtt{F}}$. The client sends a challenge to the server, in which it selects a random subset of c blocks for checking. The server runs GenProof to generate the corresponding proof, and sends it back to the client. The client then checks the validity of the received proof by running CheckProof.

The Retrieve **phase.** The Retrieve phase is activated when the client wants to replace her working copy with an older or a newer version. The client sends a request to the server. The server uses GenRetrieveVersionAndProof to generate the delta of the desired file version against the client's local version ($\delta_{retrieve}$ in Figure 6.2), together with the retrieve tag of the desired file version. Both the delta and the retrieve tag are returned to the client. The client then computes the desired file version, and checks its validity by running CheckRetrieveProof.

Let κ be a security parameter. Let $h : \{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow GF(p)$ be a PRF. All arithmetic operations are over the field $GF(p)$ of integers modulo p, where p is a large prime (at least 80 bits), unless noted otherwise explicitly. RDC-AVCS has four phases: Setup, Commit, Challenge, and Retrieve.

Setup: The client runs $(K_1, K_2) \leftarrow \mathsf{KeyGen}(1^{\kappa})$ and picks s random numbers $\alpha_1, \ldots, \alpha_s$ from $GF(p)$. The client sets $n = 0$

Commit: Having made updates to her working copy $\mathtt{F}_{t-1}$, the client C wants to commit to the repository a new version $\mathtt{F}_t$. C performs the following operations:

1. Compute δ for $\mathtt{F}_t$ against the immediate previous version $\mathtt{F}_{t-1}$, such that $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$
2. Run $(\delta_{skip}, skip(t)) \leftarrow \mathsf{ComputeDelta}(K_2, \delta, t, \mathtt{F}_t)$
3. View δ_{skip} as a collection of blocks and run $(\mathtt{T}_t, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, begin, end) \leftarrow \mathsf{GenMetadata}(K_1, K_2, \delta_{skip}, n, \alpha_1, \ldots, \alpha_s, \mathtt{F}_t, t)$. This computes a set of challenge tags $\{\mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}\}$ for the blocks in δ_{skip} and a retrieve tag $\mathtt{T}_t$ for $\mathtt{F}_t$.
4. If $(skip(t) == t - 1)$ then send $(\delta, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, \mathtt{T}_t)$ to server S; Otherwise, send $(\mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, \mathtt{T}_t)$ to S
5. Update the number of blocks in $\tilde{\mathtt{F}}$: $n = end$

Challenge: Client C uses spot checking to check possession of the virtual file $\tilde{\mathtt{F}}$. In this process, the server S uses its stored repository and the corresponding challenge tags to prove data possession.

1. C generates a challenge Q and sends Q to S. The challenge Q is a c-element set $\{(j, v_j)\}$, in which j denotes the index of the block in $\tilde{\mathtt{F}}$ to be challenged, and v_j is chosen at random from $GF(p)$.
2. S runs $(\mu_1, \ldots, \mu_s, \sigma) \leftarrow \mathsf{GenProof}(Q, \tilde{\mathtt{F}}, \mathtt{T}_1, \ldots, \mathtt{T}_n)$ and returns to C the proof of possession $(\mu_1, \ldots, \mu_s, \sigma)$
3. C checks the validity of the proof $(\mu_1, \ldots, \mu_s, \sigma)$ by running $\mathsf{CheckProof}(K_1, \alpha_1, \ldots, \alpha_s, Q, \mu_1, \ldots, \mu_s, \sigma)$

Retrieve: To replace version j (the working copy) with another version i, the client C executes:

1. C sends a request to the server S
2. The server S runs $(\delta_{retrieve}, \mathtt{T}_i) \leftarrow \mathsf{GenRetrieveVersionAndProof}(j, i)$ and returns to the client $\delta_{retrieve}$ and the retrieve tag $\mathtt{T}_i$ for version i
3. C computes $\mathtt{F}_i$: $\mathtt{F}_i = \mathtt{F}_j + \delta_{retrieve}$
4. C checks the validity of $\mathtt{F}_i$ by running $\mathsf{CheckRetrieveProof}(K_2, \mathtt{F}_i, i, \mathtt{T}_i)$

Figure 6.2 The RDC-AVCS system.

KeyGen(1^κ): Choose two keys K_1, K_2 at random from $\{0,1\}^\kappa$. Return (K_1, K_2)

ComputeDelta($K_2, \delta, t, \mathtt{F}_t$):

1. Initialize the skip delta for $\mathtt{F}_t$: $\delta_{skip} = \delta$, and run $skip(t) \leftarrow$ SelectSkipVersion(t)
2. If $(skip(t) \neq t-1)$ then client C executes:
   (a)Send $(\delta, t, skip(t))$ to the server S
   (b)The server S runs $(\delta_{reverse}, \delta_{skip}) \leftarrow$ ComputeReverseAndSkipDelta$(\delta, t, skip(t))$. S stores δ_{skip} and sends $(\delta_{reverse}, \mathtt{T}_{skip(t)})$ back to C
   (c)The client C re-computes $\mathtt{F}_{skip(t)}$: $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$. C runs CheckRetrieveProof$(K_2, \mathtt{F}_{skip(t)}, skip(t), \mathtt{T}_{skip(t)})$ to check the correctness of the $\delta_{reverse}$ received from S. If the check fails, conclude that S is faulty and exit. Otherwise, compute δ_{skip} for $\mathtt{F}_t$, such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$
3. Return $(\delta_{skip}, skip(t))$

GenMetadata($K_1, K_2, \delta, n, \alpha_1, \ldots, \alpha_s, \mathtt{F}_t, t$):

1. $begin = n+1$
2. View δ as a collection of d fixed-size blocks: $\delta = (\mathtt{b}_{n+1}, \ldots, \mathtt{b}_{n+d})$. For the purpose of computing challenge tags, we use the range $[n+1, n+d]$ for the block indices of the blocks in δ. Each block $\mathtt{b}_i$ in δ contains s symbols from $GF(p)$: $\mathtt{b}_i = (\mathtt{b}_{i,1}, \ldots, \mathtt{b}_{i,s})$.
3. $end = n+d$
4. For $begin \leq j \leq end$: $\mathtt{T}_j = h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$
5. $\mathtt{T}_t = HMAC_{K_2}(\mathtt{F}_t || t)$
6. Return $(\mathtt{T}_t, \mathtt{T}_{begin}, \ldots, \mathtt{T}_{end}, begin, end)$

GenProof($Q, \tilde{\mathtt{F}}, \mathtt{T}_1, \ldots, \mathtt{T}_n$):

1. Parse Q as a set of c pairs (j, v_j). Parse $\tilde{\mathtt{F}}$ as $\{\mathtt{b}_1, \ldots, \mathtt{b}_n\}$.
2. Compute the proof of possession $(\mu_1, \ldots, \mu_s, \sigma)$:
   •For $1 \leq k \leq s$: $\mu_k = \sum_{(j,v_j) \in Q} v_j \mathtt{b}_{jk} \mod p$
   •$\sigma = \sum_{(j,v_j) \in Q} v_j \mathtt{T}_j \mod p$
3. Return $(\mu_1, \ldots, \mu_s, \sigma)$

CheckProof($K_1, \alpha_1, \ldots, \alpha_s, Q, \mu_1, \ldots, \mu_s, \sigma$):

1. Parse Q as a set of c pairs (j, v_j)
2. If $\sigma = \sum_{(j,v_j) \in Q} v_j h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mu_k \mod p$, return "success". Otherwise return "failure".

GenRetrieveVersionAndProof(j, i):

1. Compute $\mathtt{F}_j$ by starting from $\mathtt{F}_0$ and apply the corresponding skip deltas
2. Compute $\mathtt{F}_i$ by starting from $\mathtt{F}_0$ and apply the corresponding skip deltas
3. Compute $\delta_{retrieve}$ such that $\mathtt{F}_i = \mathtt{F}_j + \delta_{retrieve}$
4. Get the retrieve tag $\mathtt{T}_i$ from the repository
5. Return $(\delta_{retrieve}, \mathtt{T}_i)$

CheckRetrieveProof($K_2, \mathtt{F}_t, t, \mathtt{T}$):

1. $\mathtt{T}_t = HMAC_{K_2}(\mathtt{F}_t || t)$
2. if $(\mathtt{T}_t == \mathtt{T})$ then return true; Otherwise, return false

Figure 6.3 The RDC-AVCS scheme.

SelectSkipVersion(t):

1. Considering the binary representation of the version number t, obtain $skip(t)$ by changing the rightmost bit that has value "1" into a bit with value "0"
2. Return $skip(t)$

ComputeReverseAndSkipDelta$(\delta, t, skip(t))$:

1. Retrieve $\mathtt{F}_t$'s immediate previous version, $\mathtt{F}_{t-1}$, based on the data in the repository
2. Compute $\mathtt{F}_t$: $\mathtt{F}_t = \mathtt{F}_{t-1} + \delta$
3. Retrieve $\mathtt{F}_{skip(t)}$ based on the data in the repository
4. Compute $\delta_{reverse}$, such that $\mathtt{F}_{skip(t)} = \mathtt{F}_t + \delta_{reverse}$
5. Compute the skip delta δ_{skip} for $\mathtt{F}_t$, such that $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$
6. Return $(\delta_{reverse}, \delta_{skip})$

Figure 6.4 Components of the RDC-AVCS scheme.

6.6 Analysis and Discussion

6.6.1 Security Analysis

The security of the RDC-AVCS scheme is captured by the following lemmas and theorems:

Lemma 6.6.1 (Corruption Detection Guarantee)**.** *Assume that the server stores an n-block file, out of which x blocks are corrupted. By randomly checking c different blocks over the entire file, the verifier (client) will detect the corruption with probability at least $1-(1-\frac{x}{n})^c$.*

Proof. We refer the reader to [7, 8] for the proof. □

Based on Lemma 6.6.1, if the server corrupts 1% of the whole file then, by randomly checking 460 blocks, the verifier can detect the corruption with a probability of at least 99%, regardless of the file size.

Lemma 6.6.2. *Let S be an RDC scheme, designed for static data, which achieves the PDP security guarantee for a file $\mathtt{F}$ outsourced at un untrusted third party [7, 8], and let S' be another RDC scheme obtained by enhancing S to support the append operation: Blocks can be appended at the end of $\mathtt{F}$ and for each appended block a verification tag is computed*

by the client and stored at the server. Then S' also achieves the PDP security guarantee for the updated file.

Proof. (sketch) We show that an RDC scheme can guarantee data possession of an updated version of the file after an arbitrary number of appends are performed. Assume the client outsources a file $\mathtt{F}$, which has n blocks $\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n$. The client applies RDC scheme S over this file as follows. During the Setup phase, it computes verification tags $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$ for all the blocks in $\mathtt{F}$. The verification tag $\mathtt{T}_i$ is computed over the data in file block $\mathtt{b}_i$ and also over i, the index of block $\mathtt{b}_i$ in $\mathtt{F}$. The client then outsources $\mathtt{F}$ as well as the verification tags to the untrusted server. During the Challenge phase, the verifier (client) uses spot checking to check the integrity of $\mathtt{F}$ [8]. This RDC scheme S guarantees data possession of file $\mathtt{F}$. We obtain a new RDC scheme S' from S by adding support for the append operation. When the client wants to append a new block $\mathtt{b}_{n+1}$ to file $\mathtt{F}$, the client computes a new verification tag $\mathtt{T}_{n+1}$ over the data in $\mathtt{b}_{n+1}$ and over the index $n+1$ of the new block. The client then sends $\mathtt{b}_{n+1}$ and $\mathtt{T}_{n+1}$ to the server. From the client's view, the server should now store the new file $\mathtt{F}'$, which has $n+1$ blocks $\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n, \mathtt{b}_{n+1}$, together with the set of tags $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n, \mathtt{T}_{n+1}$. The same argument used to prove that S achieves the PDP security guarantee over the initial file $\mathtt{F}$ can now be used to show that S' achieves the PDP security guarantee over the updated file $\mathtt{F}'$. By induction, S' can guarantee data possession of any updated version of the file after an arbitrary number of append operations are performed. Thus, we conclude that a PDP scheme which supports the append operation can achieve the PDP security guarantee for the updated file. □

Lemma 6.6.3. RDC-AVCS *guarantees that skip delta files are correctly computed by the client.*

Proof. (sketch) The skip delta may be computed in two ways during the Commit phase:

– The skip version is the version immediately previous to the new version ($skip(t) = t-1$). In this case, the client computes directly the correct skip delta.
– The skip version is not the version immediately previous to the new version ($skip(t) \neq t-1$). In this case, the client cooperates with the untrusted server to compute the skip delta. The client computes the skip version of the file based on the data received from the server and then verifies the correctness of the skip version using the retrieve tag provided by the server. This check guarantees the correctness of the skip version, since the retrieve tag was previously computed by the client. If this check is successful, the client then computes the correct skip delta.

In both cases, the skip delta is guaranteed to be correctly computed by the client. □

Lemma 6.6.3 guarantees that the client computes challenge tags over the correct skip deltas. This is important, because otherwise corruptions introduced during the commit operation may go undetected and may get incorporated in the VCS repository.

Theorem 6.6.4. RDC-AVCS *achieves security guarantees **SG1** and **SG2**.*

Proof. (sketch). In RDC-AVCS, the repository, which is the collection of t versions of file $\mathtt{F}$, can be seen as a virtual file $\tilde{\mathtt{F}}$, obtained by concatenating the initial file version $\mathtt{F}_0$, and the skip delta files δ_1, ..., δ_{t-1} corresponding to the subsequent versions. In this view, committing a new version to the repository is equivalent to appending the corresponding skip delta to the file $\tilde{\mathtt{F}}$. During the Commit phase, when committing the initial file version $\mathtt{F}_0$, the client computes the challenge tags over $\mathtt{F}_0$, and when committing each subsequent version, the client computes the challenge tags over the corresponding skip delta as if the skip delta is appended to $\tilde{\mathtt{F}}$. According to Lemma 6.6.3, each skip delta is guaranteed to be correctly computed by the client.

During the Challenge phase, the client uses spot checking to check the integrity of $\tilde{\mathtt{F}}$. RDC schemes for static data, in which there is a verification tag for each file block

have been shown to achieve the PDP security guarantee [8, 10], *i.e.*, the client can detect corruption of a fraction of the outsourced data. RDC-AVCS falls in the same category, except it supports an additional operation, append to $\tilde{\mathtt{F}}$. According to lemma 6.6.2, an RDC scheme supporting append operation achieves the same security guarantee as an RDC scheme for static data. Finally, according to lemma 6.6.1, the verifier in RDC can detect if the server corrupts a fraction of the outsourced file; thus, our RDC-AVCS scheme achieves the security guarantee **SG1**.

In RDC-AVCS, the client computes a retrieve tag for each file version $\mathtt{F}_i$ by applying an HMAC over the concatenation of the file version content ($\mathtt{F}_i$) and the version number (i) using a secret key (K_2). The security of HMAC guarantees that the adversary cannot forge a retrieve tag without knowing the secret key. Furthermore, the adversary cannot perform a replay attack by providing in the Retrieve phase a different file version than the one requested by the client. We conclude that the RDC-AVCS client can verify the correctness of the retrieved versions, thus achieving the security guarantee **SG2**. □

6.6.2 Performance Analysis

During the Commit phase, the client interacts with the sever to compute the skip deltas. To retrieve any file version from the repository, the server has to go through at most $log(t)$ skip deltas, thus, the server computation is $O(nlog(t))$. The client has to compute the skip version and the skip delta, and generate the metadata, which require a computation complexity linear in the version size (see Table 3.1). The communication in a commit operation is also linear with the version size, since it mainly includes two deltas (Figure 6.2 and 6.3) and a set of challenge tags for a skip delta.

During the Challenge phase, RDC-AVCS adopts the spot checking technique, in which the client challenges the server to prove possession of a random subset of the blocks in $\tilde{\mathrm{F}}$ (the number of challenged blocks is always a small constant [7]), and the server generates a proof of data possession by aggregating the selected blocks and the corresponding challenge tags. Thus, the computation (client and server) and the communication complexity are both $O(1)$ (Table 3.1). This is a major advantage of RDC-AVCS compared to previous schemes, in which the checking complexity is determined either by the repository size or the version size (Table 3.1).

During the Retrieve phase, to retrieve a version from the repository, the server needs to apply at most $log(t)$ skip deltas, thus, the server computation is $O(nlog(t))$. Previous schemes which are built on top of delta encoding (or can be easily built on top of delta encoding) impose $O(nt)$ computation on the server (Table 3.1). The client storage overhead in RDC-AVCS is $O(n)$, since the client always stores locally the working copy.

6.6.3 Remarks

Small corruption protection. In RDC-AVCS, we adopt spot checking during the Challenge phase for efficiency reasons. Spot checking was shown to detect data corruption with high probability if the server corrupts a fraction of the data [7]. This provides defense against an adversary which is rational and economically motivated, *i.e.*, one that will not cheat unless it can achieve a clear financial gain without being detected. Unfortunately, spot checking is not necessarily effective under a stronger adversary, *e.g.*, an adversary which is fully malicious. Spot checking cannot detect if the adversary corrupts a small amount of the data, such as 1 byte. To provide protection against small amounts of data corruption – a property called *robustness* – previous RDC schemes for static data rely on a special

application of error correcting codes to generate redundant data, so that small corruptions that are not detected can be repaired [8, 17, 21]. Integrating error correcting codes with RDC when dynamic updates can be performed on the data is much more challenging than in the static setting. A few RDC solutions have been proposed to achieve robustness for the dynamic setting, but this involves substantial additional cost: one system requires to store a large amount of redundant data on the client side [15]; other systems store and access the redundant data on the server side either by requiring the client to access the entire redundancy [24] or by using inefficient mechanisms such as PIR that hide the access pattern [36].

In this work, we choose to sacrifice robustness for two reasons. First, the solutions proposed to achieve robustness for RDC under a dynamic setting are designed to handle the full range of update operations (insertions, deletions, modifications) and are thus overkill for version control systems where the only meaningful operation is append. Second, one of our main design goals was to achieve an auditable VCS scheme which is efficient and has performance comparable to a regular (non-secure) VCS system.

Multiple-file support. We have described RDC-AVCS for the case when the main repository only contains the versions of one file. A challenge tag for block with index j in $\tilde{\mathtt{F}}$ is computed as $\mathtt{T}_j = h_{K_1}(j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$. The index j used in the challenge tag should be different across all the challenge tags. In other words, the client should not reuse the same index j twice for computing challenge tags. In this case, the index j used in the challenge tag is the block's position in the file $\tilde{\mathtt{F}}$, which ensures its unicity. When multiple files are stored in the VCS repository, the client must ensure that the indices used to compute the challenge tags are different not only across blocks of the same file, but also

across blocks of different files. This could be achieved by prepending a file identifier to the block index. For example, if the identifier of a file F is given by $id(\mathtt{F})$ and assuming that each file has a unique identifier, then for the blocks in the various versions of F, the client computes challenge tags as $\mathtt{T}_j = h_{K_1}(id(\mathtt{F})||j) + \sum_{k=1}^{s} \alpha_k \mathtt{b}_{jk}$. Similarly, the file's identifier should be embedded in the retrieve tag for version $\mathtt{F}_i$: $\mathtt{T}_i = h_{K_2}(\mathtt{F}_i||id(\mathtt{F})||i)$.

6.7 Implementation and Experiments

6.7.1 Implementation

We built a prototype for RDC-AVCS on top of Apache Subversion (SVN) [85], a popular open-source version control system. We added about 4,000 lines of C code into the SVN code base (V1.7.8), and built Secure SVN (SSVN), a secure version control system based on skip delta encoding. Since many SVN repositories already exist, we also built a tool, SSVN-Migrate, which converts an existing (non-secure) SVN repository into a SSVN repository.

Implementation overview. We modified the source code in both SVN client and SVN server. For the SVN client, we mainly modified the following SVN commands

- svn add: add files to the working copy. The corresponding new command in SSVN is "ssvn add".
- svn rm: remove files from the working copy. The corresponding new command in SSVN is "ssvn rm".
- svn commit: commit the changes to the repository. The corresponding new command in SSVN is "ssvn commit".
- svn co: checkout the latest version of the data. The corresponding new command in SSVN is "ssvn co".
- svn update: update the current version to an arbitrary version. The corresponding new command in SSVN is "ssvn update".

For the SVN server, we modified the stand-alone server "svnserve". The new server is named "sec-svnserve".

During the Commit phase, the client updates the working copy and wants to commit the changes to the repository. In RDC-AVCS, the changes for a new version $\mathtt{F}_t$ are encoded in the skip delta, δ_{skip}, which is the difference between the skip version and the new version, *i.e.*, $\mathtt{F}_t = \mathtt{F}_{skip(t)} + \delta_{skip}$. The algorithm for computing δ_{skip} is described in Section 6.5.3. After computing the skip delta, the client computes the set of challenge tags for it and a retrieve tag for the new version, and sends them to the server.

In SSVN we added functionality to the original SVN client ("svn commit"), so that the SSVN client ("ssvn commit") can communicate with the server to compute the skip delta, as well as compute the challenge and retrieve tags. We also added functionality to the original SVN server ("svnserve") to allow the server to compute and send back the delta of skip version against the new version, together with a proof for checking the validity of the skip version.

During the Retrieve phase, the client wants to revert the working copy to an older version or update it to a newer version. It sends a request to the server, which retrieves the requested version from the repository, together with the corresponding retrieve tag. The server can then choose to send back either the whole requested version or the delta between the requested version and the working copy (SVN uses the latter strategy). The client further validates the requested version based on the retrieve tag. Correspondingly, in SSVN we added additional functionality to the original SVN client ("svn co" and "svn update"), so that the SSVN client ("ssvn co" and "ssvn update") can verify the retrieve tags for the affected files. We also added additional functionality to the original SVN server ("svnserve") to allow it to retrieve and send back the corresponding retrieve tags for the affected files.

Implementation issues. We highlight next some of the most interesting implementation issues we encountered. First, we had to bridge the gap between how RDC-AVCS and SVN view the data: RDC-AVCS abstracts each version of the data as a file, and thus one simply performs update operations to this file. However, in SVN, each version is associated with a project, which is a collection of files, and the delta (*i.e.*, skip delta) is computed independently for each file. In addition, files can be added and deleted from the project. To reconcile the different views, we apply RDC-AVCS over each file in an SVN project, *i.e.*, we have a virtual project for each file, and the SVN project is a collection of virtual projects corresponding to the files in the SVN project. When a file is added to the project, the corresponding virtual project is initialized; when this file is updated (*i.e.*, insert, delete, modify, or append data), the corresponding virtual project is updated; when the file is deleted, the corresponding virtual project should be kept rather than be deleted.

Another implementation issue is related to how SVN handles memory management. Rather than requesting memory directly from the OS using the standard $malloc()$ function, SVN relies on Apache Portable Runtime (APR) [105] library for memory management. Specifically, a program that links against APR can request a pool of memory by using $apr\_pool\_create()$, and APR will allocate a moderate-size chunk of memory from the OS which will be available for use to the program immediately. The pool will automatically grow in size to accommodate programs that request more memory than the original pool contained. Unfortunately, without carefully reclaiming back memory from the pool when handling a large number of files, the pool becomes full, leading to an "out of memory" error. In SSVN, we tackled this issue by clearing the pool after having handled a certain number of files, *e.g.*, 1000. We tested that SSVN is robust enough to handle hundreds of thousands of files in a single commit operation.

6.7.2 Experimental Setup

We ran experiments in which both the server and the client are running on the same machine, an Intel Core 2 Duo system with two CPUs (each running at 3.0GHz, with a 6144KB cache), 1.333GHz frontside bus, 4GB RAM and a Hitachi HDP725032GLA360 360GB hard disk with ext4 file system. The system runs Ubuntu 12.10, kernel version 3.5.0-17-generic. We used the OpenSSL library [44] version 1.0.1e.

Repository selection. We categorized the existing SVN repositories into three groups based on the number of files in the repository: A small-size repository has less than $5,000$ files, a medium-size repository has between $5,000$ and $50,000$ files, and a large-size repository has more than $50,000$ files. Based on these criteria, we selected three representative public SVN repositories for our experimental evaluation: FileZilla [106] for small-size repository, Wireshark [107] for medium-size repository, and GCC [96] for large-size repository. Table 6.2 shows statistics about these three repositories.

Table 6.2 Statistics for The Selected Repositories of June 2013

	FileZilla	Wireshark	GCC
Dates of activity	2001-2013	1998-2013	1987-2013
Number of versions	5,119	49,946	200,127
Number of files	1,023	5,342	80,183
Average filesize	19KB	32KB	6KB
Repository category	small size	medium size	large size

Overview of experiments. We evaluated the computation and communication overhead during the Commit phase (Section 6.7.3) and the computation overhead during the Retrieve phase (Section 6.7.4), for both SSVN and SVN. The Challenge phase has been shown to be very efficient for RDC schemes which rely on spot checking [7], so we do not include it in our experiments.

We average the overhead over the first 1000 versions of the three repositories (labeled FileZilla, Wireshark and GCC1). GCC has a large-size repository, with more than 200K versions and more than 80K files in its latest version. Since for GCC the difference between the first 1000 versions and the last 1000 versions is considerable in the size of the repository, we also included in our experiments an average of the overhead over the last 1000 versions of GCC (labeled GCC2).

In Section 6.7.5, we describe the migration tool which seamlessly converts an existing (non-secure) SVN repository to a SSVN repository; we also perform an experiment in which we migrate the first 3000 versions of the aforementioned three repositories.

6.7.3 Commit Phase

For SVN and SSVN, we evaluated the computation and communication overhead for the commit operation. To measure the time for a commit operation, we measured the time needed for running the shell commands "svn commit" and "ssvn commit" to commit a version. To measure the communication overhead of non-secure SVN for a commit operation, we observed that the non-secure SVN client relies on two write functions $writebuf\_output$ and $svn\_ra\_svn\_writebuf\_output$ to send data, and two read functions $readbuf\_input$ and $svn\_ra\_svn\_readbuf\_input$ to receive data. Thus, for each commit operation, we accumulate the data sent in the write functions, which are the total communication from the client to the server. Similarly, we accumulated the data received in the read functions, which are the total communication from the server to the client. SSVN also relies on these four I/O functions, thus we measured its communication overhead similarly.

The experimental results for the commit phase are shown in Tables 6.3, 6.4 and 6.5. We have several observations: Firstly, compared to the non-secure SVN, SSVN adds only a small overhead to the total computation (between 3% and 11% in Table 6.3) and the total communication from the client to the server (between 3% and 7% in Table 6.4). Secondly, SSVN adds more overhead to the communication from the server to the client because in SSVN the client retrieves data from the server to facilitate the computation of skip deltas during commit; in contrast, for non-secure SVN, the client does not need to compute the skip deltas locally and the server only sends back small control messages. This is the main cost we need to pay for offering a secure version of SVN. Although the communication overhead in Table 6.5 is higher for SSVN, we note that in the worst case the additional overhead for committing one version in GCC2 is less than 3KB.

Table 6.3 The Average Time for Committing One Version in Both SSVN And Non-secure SVN

	FileZilla	Wireshark	GCC1	GCC2
SSVN (s)	0.427	0.416	0.417	10.776
non-secure SVN (s)	0.389	0.376	0.386	10.502

Table 6.4 The Average Communication from The Client to The Server for Committing One Version in Both SSVN And Non-secure SVN

	FileZilla	Wireshark	GCC1	GCC2
SSVN (KB)	4.599	3.458	4.123	6
non-secure SVN (KB)	4.391	3.246	4.017	5.696

Table 6.5 The Average Communication from The Server to The Client for Committing One Version in Both SSVN And Non-secure SVN

	FileZilla	Wireshark	GCC1	GCC2
SSVN (KB)	1.559	1.437	1.047	3.244
non-secure SVN (KB)	0.574	0.58	0.574	0.571

6.7.4 Retrieve Phase

For SSVN and non-secure SVN, we evaluated the computation overhead for the retrieve operation by measuring the time needed to run the shell commands "svn update -r i" (for non-secure SVN) and "ssvn update -r i" (for SSVN) to retrieve a version i by updating version $i-1$. The corresponding experimental results are shown in Table 6.6. We observe that, compared to non-secure SVN, SSVN adds a reasonable overhead: Table 6.6 shows the time needed to retrieve a version in SSVN increases between 6% and 29% compared to non-secure SVN. Note that this additional time is less than 0.3 seconds in the worst case (for GCC2). The additional overhead is caused by checking the validity of the corresponding version, *i.e.* re-computing the retrieve tags for the affected files in this version and comparing them with the retrieve tags sent back by the server. We did not provide evaluation for communication overhead, since there is no additional communication from the client to the server, and the additional communication from the server to the client will only contains retrieve tags of the affected files in this version (we use HMAC-SHA1 to implement retrieve tags, so only 20 bytes are needed for one retrieve tag).

Table 6.6 The Average Time for Retrieving One Version in Both Secure And Non-secure SVN

	FileZilla	Wireshark	GCC1	GCC2
secure SVN (s)	0.0535	0.0453	0.0506	5.086
non-secure SVN (s)	0.0416	0.0376	0.0416	4.779

6.7.5 Migrating Repositories from Non-Secure SVN to SSVN

Many commercial and non-commercial projects are using SVN for source control management (*e.g.*, FreeBSD [108], GCC, Wireshark, all the open-source projects in Apache Software Foundation [109], etc.). Such projects already have repositories created based on non-secure SVN. To facilitate the migration from non-secure SVN to SSVN,

we built **SSVN-Migrate**, a tool that seamlessly converts an existing non-secure SVN repository into a repository for SSVN. SSVN-Migrate works as follows: Starting from the first version (*i.e.*, an empty version), each time it calls "svn update" to check out a new version of the data from the non-secure SVN repository (*i.e.*, version number increased by 1), uses "ssvn add" and "ssvn rm" to update the working copy, and then calls "ssvn commit" to commit the changes into the SSVN repository.

We used SSVN-Migrate to migrate FileZilla, Wireshark and GCC to secure SVN. Table 6.7 shows the time needed for migrating all the first 3000 versions of these SVN repositories. We observe that the time needed for migrating the same collection of versions from different SVN repositories does not vary a lot. One possible reason is that the migration time is mainly determined by the repository size, which is approximately linear to the version number.

Note that our SSVN-Migrate tool tries to re-use as much as possible components we have built for SSVN or existing SVN commands. We believe the results can be significantly improved by optimizing the migration process (*e.g.*, work directly with the raw non-secure and secure repositories), using more powerful hardware, or obtaining additional computing resources from public cloud computing services.

Table 6.7 The Time for Migrating The First 3000 Versions of The Existing SVN Repositories to SSVN

	FileZilla	Wireshark	GCC
total time (s)	1,934	1,909	1,719

CHAPTER 7

CONCLUSION

Traditionally, data owners can only store and manage their data in their own data centers which, unfortunately, may incur a considerable financial burden. The emerging cloud storage model allows data owners to outsource the storage of their data to Cloud Storage Providers (CSPs) and achieve several benefits such as low cost, improved reliability and scalability, and great flexibility. However, data owners today may be reluctant to outsource their data, simply because they do not trust the CSPs, and they are not convinced that their outsourced data will be adequately protected and maintained over time. In this dissertation, several remote data checking protocols have been designed and implemented in order to ensure the integrity and retrievability of the outsourced data: R-DPDP, RDC-NC, RDC-SR, ERDC-SR and RDC-AVCS.

Adding protection against small corruptions to remote data checking schemes that support dynamic updates extends the range of applications that rely on outsourcing data at untrusted servers. We design Robust Dynamic Provable Data Possession (R-DPDP) schemes that provide robustness and, at the same time, support dynamic data updates, while requiring small, constant, client storage. The main challenge that had to be overcome was to reduce the client-server communication overhead during updates under an adversarial setting.

The second RDC protocol presented is RDC-NC, a novel distributed RDC scheme for network coding-based distributed storage systems that rely on untrusted servers. Our RDC-NC scheme can be used to ensure data remains intact when faced with data corruption,

replay, and pollution attacks. We built a prototype for RDC-NC and experimentally evaluated its performance, showing that RDC-NC is inexpensive for both clients and servers.

Having observed that all the existing distributed RDC schemes involve the client heavily during the Repair phase, we investigate a new self-repairing concept, in which the untrusted servers are responsible to repair the corruption, while the client acts as a lightweight coordinator during repair. Based on this novel concept, we design two schemes for replication-based distributed storage systems, RDC-SR and ERDC-SR. They 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 an analytical performance analysis (for ERDC-SR).

All the aforementioned RDC protocols can only ensure the integrity of the latest version of the outsourced data. Ensuring the integrity of a version control system (VCS) may provide data owners additional benefits, *e.g.*, allowing them to roll back to an old file version when the working file version is corrupted. We introduce Auditable Version Control Systems (AVCS), which are delta-based VCS systems designed to function under an adversarial setting. We propose RDC-AVCS, an AVCS scheme for skip delta-based version control systems, which relies on RDC mechanisms to ensure all the versions of a file can be retrieved from the untrusted VCS server over time. Unlike previous solutions which rely on dynamic RDC and are interesting from a theoretical point of view, our RDC-AVCS scheme is the first pragmatic approach for auditing real-world VCS systems. Our security analysis

and experimental evaluation show that RDC-AVCS achieves the desired security guarantees at the cost of a modest decrease in performance compared to a regular (non-secure) VCS system.

APPENDIX A

DYNAMIC PROVABLE DATA POSSESSION

A.1 Definition of Dynamic Provable Data Possession

A complete definition of a DPDP scheme [12] is provided in the following.

Definition A.1.1. (DPDP SCHEME) *A Dynamic Provable Data Possession (DPDP) scheme is a collection of seven polynomial-time algorithms:*

- KeyGen_DPDP$(1^{\kappa}) \rightarrow \{sk, pk\}$*: a probabilistic key generation algorithm run by the* ***client*** *to setup the scheme.* Input: *the security parameter* κ*.* Output: *the secret key* sk *and public key* pk*.*
- PrepareUpdate_DPDP$(sk, pk, F, info, M_c) \rightarrow \{e(F), e(info), e(M)\}$*: an algorithm run by the* ***client*** *to prepare (a part of) the file for untrusted storage.* Input: *the secret key* sk*, the public key* pk*, (a part of) the file* F*, information about the update operation* $info$ *(*e.g.*, full re-write, modify block* i*, delete block* i*, insert a block after block* i*, etc), and the previous metadata* M_c*.* Output: *the "encoded" version of (a part of) the file* $e(F)$ *(by adding randomness, adding sentinels, etc.), the information about the update operation* $e(info)$ *(changed to fit the encoded version), and the new metadata* $e(M)$*. The client will send* $e(F), e(info)$*, and* $e(M)$ *to the server.*
- PerformUpdate_DPDP$(pk, F_{i-1}, M_{i-1}, e(F), e(info), e(M) \rightarrow \{F_i, M_i, M_c^{'}, P_{M_c^{'}}\}$*: an algorithm run by the* ***server*** *in response to an update request from the client.* Input: *public key* pk*, the old version of the file* F_{i-1}*, the metadata* M_{i-1}*, and the values* $e(F), e(info)$ $e(M)$ *provided by the client.* Output: *the new version of the file* F_i *and metadata* M_i*, the metadata to be sent to client* $M_c^{'}$ *and its proof of correctness* $P_{M_c^{'}}$*. The server will send* $M_c^{'}$ *and* $P_{M_c^{'}}$ *back to the client.*
- VerifyUpdate_DPDP$(sk, pk, F, info, M_c, M_c^{'}, P_{M_c^{'}}) \rightarrow \{$accept, reject$\}$*: an algorithm run by the* ***client*** *to verify the server's behavior during the update.* Input: *all the inputs from* PrepareUpdate_DPDP *and the values* $M_c^{'}$*,* $P_{M_c^{'}}$ *which are sent by the server.* Output: accept *if the verification succeeds,* reject *otherwise.*
- GenChallenge_DPDP$(sk, pk, M_c) \rightarrow \{c\}$*: a probabilistic algorithm run by the* ***client*** *to issue a challenge for the server.* Input: *the secret key* sk*, public key* pk*, and the latest client metadata* M_c*.* Output: *the challenge* c *that will be sent to the server.*

- Prove_DPDP$(pk, F_i, M_i, c) \rightarrow \{\Pi\}$*: an algorithm run by the* ***server*** *to generate the proof of possession upon receiving the challenge from the client.* Input: *the public key* pk*, the latest version of file* F_i*, the metadata* M_i*, and the challenge* c*.* Output: *a proof of possession* Π *that will be sent back to the client.*

- Verify_DPDP$(sk, pk, M_c, c, \Pi) \rightarrow \{\text{accept}, \text{reject}\}$*: an algorithm run by the* ***client*** *to validate a proof of possession upon receiving the proof* Π *from the server.* Input: *the secret key* sk*, the public key* pk*, the client metadata* M_c*, the challenge* c*, and the proof* Π*.* Output: accept *if* Π *is a valid proof of possession,* reject *otherwise.*

APPENDIX B

REPLAY ATTACKS IN NETWORK CODING-BASED DISTRIBUTED STORAGE SYSTEMS

B.1 Replay Attack against A Basic Network Coding-based Scheme

Assume the following chain of events in a configuration similar with the one in Figure 3.1(c), in which the attacker can corrupt at most one server (out of three) in each epoch.

Epoch 1: Each server stores 2 network coded blocks.

$$S_1 \ : \ \mathtt{x}_{11} = \mathtt{b}_1 \quad , \quad \mathtt{x}_{12} = \mathtt{b}_2 + \mathtt{b}_3$$

(*i.e.*, S_1 stores coded blocks $\mathtt{x}_{11}, \mathtt{x}_{12}$)

$$S_2 \ : \ \mathtt{x}_{21} = \mathtt{b}_3 \quad , \quad \mathtt{x}_{22} = \mathtt{b}_1 + \mathtt{b}_2$$

$$S_3 \ : \ \mathtt{x}_{31} = \mathtt{b}_1 + \mathtt{b}_3 \quad , \quad \mathtt{x}_{32} = \mathtt{b}_2 + \mathtt{b}_3$$

The attacker corrupts S_3, but keeps a copy of $\mathtt{x}_{31}, \mathtt{x}_{32}$ (and their challenge tags).

The client detects corruption at S_3. Thus, it contacts S_1 and S_2 to generate new blocks

$$\mathtt{x}'_{31} = 1 \cdot (1 \cdot \mathtt{x}_{11} + 1 \cdot \mathtt{x}_{12}) + 1 \cdot (1 \cdot \mathtt{x}_{21} + 0 \cdot \mathtt{x}_{22}) = \mathtt{b}_1 + \mathtt{b}_2 + 2\mathtt{b}_3$$

$$\mathtt{x}'_{32} = 1 \cdot (1 \cdot \mathtt{x}_{11} + 0 \cdot \mathtt{x}_{12}) + 1 \cdot (0 \cdot \mathtt{x}_{21} + 1 \cdot \mathtt{x}_{22}) = 2\mathtt{b}_1 + \mathtt{b}_2$$

The new blocks $\mathtt{x}'_{31}$ and $\mathtt{x}'_{32}$ are then stored at S_3.

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

Epoch 2: The attacker corrupts S_1. The client detects corruption at S_1; thus, it contacts $S2$ and S_3 to generate new blocks.

$$\mathtt{x}'_{11} = 1 \cdot (1 \cdot \mathtt{x}_{21} - 4 \cdot \mathtt{x}_{22}) + 1 \cdot (1 \cdot \mathtt{x}'_{31} + 3 \cdot \mathtt{x}'_{32}) = 3\mathtt{b}_1 + 3\mathtt{b}_3$$

$$\mathtt{x}'_{12} = 1 \cdot (1 \cdot \mathtt{x}_{21} + 5 \cdot \mathtt{x}_{22}) + 1 \cdot (1 \cdot \mathtt{x}'_{31} - 3 \cdot \mathtt{x}'_{32}) = 3\mathtt{b}_2 + 3\mathtt{b}_3$$

The new blocks $\mathtt{x}'_{11}$ and $\mathtt{x}'_{12}$ are then stored at S_1.

The current configuration is now:

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

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

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

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

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

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

B.2 A Simulation to Validate Theorem 3.4.1

To validate Theorem 3.4.1 experimentally, a simulation is performed as follows: For each of the first 10,000 epochs, the attacker corrupts the blocks on $n - k$ servers, forcing the

data owner to repair the blocks on the corrupted servers. This strategy gives the attacker the chance to accumulate new sets of coefficient vectors (encrypted) and the corresponding blocks. The attacker reuses the accumulated data over the next 10,000 epochs in an attempt to execute replay attacks. Since the coefficients are encrypted, the attacker has no better strategy than picking at random from the accumulated coefficient vectors.

The simulations show that encrypting the coefficients successfully mitigates replay attacks. The reason is that accumulating previously encrypted coefficient vectors does not increase the chances of performing a successful replay attack. For various combinations of (n, k) (shown in Table B.1), the attacker is unable to execute not even one successful replay attack during the 10,000 attempts.

Table B.1 Test Cases for Simulating The Replay Attack

n	k	m	α	ℓ	β
10	5	15	5	5	1
15	5	15	5	5	1
20	5	15	5	5	1
25	5	15	5	5	1
15	10	55	10	10	1
20	10	55	10	10	1
20	15	120	15	15	1

APPENDIX C

EXPERIMENTS ON THE AMAZON CLOUD

C.1 Measurements for The Amazon CSP

Tables C.1 and C.2 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 [110] to download a large file. To measure the propagation delay, we adopt the method introduced in [63] 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 C.1 and C.2 are averaged over 20 runs.

Table C.1 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 C.2 Propagation Delay (in Milliseconds)

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

C.2 Sampling Blocks from Amazon S3

We wrote a program running in an EC2 instance (Amazon Virginia region) to randomly sample 4KB blocks from S3 Virginia region. We collect the time in Table C.3. All the results are averaged over 20 runs.

Table C.3 The Time for Randomly Sampling 4KB Blocks from S3 Virginia Region

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

APPENDIX D

MULTIPLE QUANTIFICATIONS IN ERDC-SR

D.1 Quantify The Computation Required for Generating One Replica Block from The Original File Blocks in ERDC-SR

From Figure 5.2, we want to generate a block in level $log\beta$ from the original file blocks which are in level 0. Each block in level $log\beta$ depends on 2 blocks in level $log\beta-1$, *i.e.*, we need 1 cryptographic transformation in order to generate this block by knowing the 2 blocks in level $log\beta-1$. Similarly, each of the 2 blocks in level $log\beta-1$ depends on a different set of 2 blocks in level $log\beta-2$, *i.e.*, by knowing the 4 blocks in level $log\beta-2$, we need 2 cryptographic transformations in order to generate the 2 blocks in 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 in 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.

D.2 Quantify The Computation Needed to Generate A Replica Block When The Adversary Only Stores One Intermediate Block

In Figure 5.2, there are $log\beta+1$ levels, and n blocks at each level. Let i denote the level index, *i.e.*, $0 \leq i \leq log\beta$. Assume the adversary only stores one block at this server, which can be the original file block ($i=0$), the intermediate block ($0<i<log\beta$), or the block from the corresponding replica (replica block, $i=log\beta$). We want to quantify the overall

computation needed to generate a replica block being challenged, considering the adversary only stores one block in level i, and is able to access the original file (Section 4.3.2).

Considering a block from level i (we denote this block as $\mathscr{B}$) is stored by the adversary, we can categorize the replica blocks as two types: blocks which are related to $\mathscr{B}$ (type I), and blocks which are not related to $\mathscr{B}$ (type II). According to Figure 5.2, $\mathscr{B}$ will be related to $2^{log\beta-i}$ replica blocks, *e.g.*, $\mathscr{B}$ will be related to $2^{log\beta}(=\beta)$ replica blocks if it is from level 0 (*i.e.*, $i=0$), and will be related to $2^{log\beta-1}(=\frac{\beta}{2})$ replica blocks if it is from level 1 (*i.e.*, i=1). Thus, the number of replica blocks belonging to type I is $2^{log\beta-i}$, and the number of replica blocks belonging to type II is $n-2^{log\beta-i}$. If the challenged replica block belongs to type II, to compute such a block, the adversary needs to start from the original file blocks, and the computation is $\beta-1$ cryptographic transformations (see Appendix D.1). Otherwise, if the challenged replica block belongs to type I, to compute such a block, the adversary does not need to compute the intermediate block $\mathscr{B}$ as well as all the other intermediate blocks needed to compute $\mathscr{B}$. Thus, the computation needed to generate such a replica block is $\beta-1-(2^i-1)(=\beta-2^i)$ cryptographic transformations. During Challenge, the replica blocks being challenged are picked randomly, thus, 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 from type I category is $\frac{2^{log\beta-i}}{n}$, and the probability that it is from type II category 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.

D.3 Determine 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 transformation (see Appendix D.1). To generate all the $c \cdot (1-\alpha)$ missing challenged blocks, the α-cheating server at least needs to perform $n + \frac{1}{2} \cdot c \cdot (1-\alpha) - 2$ cryptographic transformation based on the following observations: the adversary needs $n-1$ cryptographic transformation to generate the first missing block; to generate the remaining $c \cdot (1-\alpha) - 1$ missing blocks, the adversary at least needs $\frac{c \cdot (1-\alpha) - 2}{2}$ cryptographic transformation 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 transformation. 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.*, $e > \frac{((1+\rho)^{\phi} - 1) \cdot \tau}{n + \frac{1}{2} \cdot c \cdot (1-\alpha) - 2}$.

D.4 Quantify The Probability That All The $c \cdot (1-\alpha)$ Missing Challenged Blocks Depend on Different Sets of β Original File Blocks

From Figure 5.2, each of the replica blocks depends on a set of β original file blocks, *e.g.*, a replica block with index i depends on a set of β file blocks with indices in the range

$[\lfloor \frac{i}{\beta} \rfloor \cdot \beta, (\lfloor \frac{i}{\beta} \rfloor + 1) \cdot \beta - 1]$, where $0 \leq i \leq n-1$ (Section 5.3.2). 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 the probability of E (*i.e.*, $P(E)$) as follows.

We observed from Figure 5.2 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 $0 \leq i \leq n-1$. We define the collection of replica blocks from this same range as a dependency group, *i.e.*, totally we have $\frac{n}{\beta}$ 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 j-th missing challenged block ($1 \leq j \leq c \cdot (1-\alpha)$) is from a different dependency group than the previously missing challenged blocks. Thus, $P(E) = \Pi_{j=1}^{c \cdot (1-\alpha)} P(E_j)$. We further evaluate $P(E_j)$ in the following:

$P(E_1) = 1$

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

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

...

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

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

D.5 Quantify 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, in Figure 5.2, 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 $0 \leq i \leq n-1$. 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 $1 \leq x \leq c \cdot (1-\alpha)$. 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 at least needs to generate x blocks purely from the original file (*i.e.*, cannot re-use any intermediate block), and generating a block purely from the original file requires $\beta - 1$ cryptographic transformation (Appendix D.1). 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 transformation.

Computing all the $P(E_x)$ values (where $1 \leq x \leq c \cdot (1-\alpha)$) will be quite complicate, thus, rather than compute 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 when $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$, $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1}) > \cdots > P(E_1)$, based on the following analysis. For the base case, we have $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1})$, because: From Appendix D.4, $P(E_{c\cdot(1-\alpha)}) = \Pi_{j=1}^{c\cdot(1-\alpha)-1} \frac{n-j\beta}{n-j}$. Similarly, we can compute $P(E_{c\cdot(1-\alpha)-1}) = (\Pi_{j=1}^{c\cdot(1-\alpha)-2} \frac{n-j\beta}{n-j}) \cdot \frac{(c\cdot(1-\alpha)-1)\cdot(\beta-1)}{n-(c\cdot(1-\alpha)-1)}$. When $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$, $n - (c \cdot (1-\alpha) - 1) \cdot \beta > c \cdot (1-\alpha) \cdot \beta > 0$. Since $0 < (c \cdot (1-\alpha) - 1) \cdot \beta < c \cdot (1-\alpha) \cdot \beta$, $\frac{P(E_{c\cdot(1-\alpha)})}{P(E_{c\cdot(1-\alpha)-1})} = \frac{n-(c\cdot(1-\alpha)-1)\cdot\beta}{(c\cdot(1-\alpha)-1)(\beta-1)} > 1$, *i.e.*, $P(E_{c\cdot(1-\alpha)}) > P(E_{c\cdot(1-\alpha)-1})$. For the general case, we have $P(E_{x+1}) > P(E_x)$, where $1 \leq x \leq c \cdot (1-\alpha) - 1$, because: 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 from the whole $\frac{n}{\beta}$ 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 $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$, we have $\frac{n}{\beta} > 2 \cdot c \cdot (1-\alpha)$. Since $x \leq c \cdot (1-\alpha) - 1$ and $\frac{n}{\beta} > 2 \cdot c \cdot (1-\alpha)$, we have $\frac{n}{\beta} - x > c \cdot (1-\alpha)$, *i.e.*, $\frac{n}{\beta} - x > x$. Considering both $\frac{n}{\beta} - x > x$ 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 aforementioned observation, we have Theorem D.5.1.

Theorem D.5.1. *When $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$, 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 inequality (D.1) always holds when $\frac{1}{2} \cdot c \cdot (1-\alpha) < x < c \cdot (1-\alpha)$ and $n \geq 2 \cdot c \cdot (1-\alpha) \cdot \beta$.

$$P(E_x) \cdot x + P(E_{c \cdot (1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) > \frac{1}{2}(P(E_x) + P(E_{c \cdot (1-\alpha)-x})) \cdot (c \cdot (1-\alpha)) \quad \text{(D.1)}$$

We justify inequality (D.1) by showing $P(E_x) \cdot x + P(E_{c \cdot (1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) - \frac{1}{2}(P(E_x) + P(E_{c \cdot (1-\alpha)-x})) \cdot (c \cdot (1-\alpha)) > 0$ in the following.

$$P(E_x) \cdot x + P(E_{c \cdot (1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) - \tfrac{1}{2}(P(E_x) + P(E_{c \cdot (1-\alpha)-x})) \cdot (c \cdot (1-\alpha))$$
$$= P(E_x) \cdot (c \cdot (1-\alpha) - (c \cdot (1-\alpha) - x)) + P(E_{c \cdot (1-\alpha)-x}) \cdot (c \cdot (1-\alpha) - x) - \tfrac{1}{2}(P(E_x) +$$

$P(E_{c\cdot(1-\alpha)-x})) \cdot (c \cdot (1-\alpha))$

$= \frac{1}{2}\cdot P(E_x)\cdot c\cdot(1-\alpha) - \frac{1}{2}\cdot P(E_{c\cdot(1-\alpha)-x})\cdot c\cdot(1-\alpha) - (P(E_x) - P(E_{c\cdot(1-\alpha)-x}))\cdot(c\cdot(1-\alpha)-x)$

$= (P(E_x) - P(E_{c\cdot(1-\alpha)-x}))(x - \frac{1}{2}\cdot c\cdot(1-\alpha))$

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 $(P(E_x) - P(E_{c\cdot(1-\alpha)-x}))(x - \frac{1}{2}\cdot c\cdot(1-\alpha)) > 0$, 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 D.1, we prove the theorem by differentiating two cases: $c\cdot(1-\alpha)$ is odd and $c\cdot(1-\alpha)$ is even.

- $c\cdot(1-\alpha)$ is odd: $\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x = P(E_{c\cdot(1-\alpha)-1})\cdot(c\cdot(1-\alpha)-1) + P(E_1)\cdot 1 + P(E_{c\cdot(1-\alpha)-2})\cdot(c\cdot(1-\alpha)-2) + P(E_2)\cdot 2 + \cdots + P(E_{\lceil\frac{1}{2}\cdot c\cdot(1-\alpha)\rceil})\cdot(\lceil\frac{1}{2}\cdot c\cdot(1-\alpha)\rceil) + P(E_{\lfloor\frac{1}{2}\cdot c\cdot(1-\alpha)\rfloor})\cdot(\lfloor\frac{1}{2}\cdot c\cdot(1-\alpha)\rfloor) > \frac{1}{2}(P(E_{c\cdot(1-\alpha)-1}) + P(E_1))\cdot(c\cdot(1-\alpha)) + \frac{1}{2}(P(E_{c\cdot(1-\alpha)-2}) + P(E_2))\cdot(c\cdot(1-\alpha)) + \cdots + \frac{1}{2}(P(E_{\lceil\frac{1}{2}\cdot c\cdot(1-\alpha)\rceil}) + P(E_{\lceil\frac{1}{2}\cdot c\cdot(1-\alpha)\rceil}))\cdot(c\cdot(1-\alpha)) = \frac{1}{2}\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)(c\cdot(1-\alpha)) = \frac{1}{2}\cdot(1 - P(E_{c\cdot(1-\alpha)}))(c\cdot(1-\alpha))$.
- $c\cdot(1-\alpha)$ is even: $\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x = P(E_{c\cdot(1-\alpha)-1})\cdot(c\cdot(1-\alpha)-1) + P(E_1)\cdot 1 + P(E_{c\cdot(1-\alpha)-2})\cdot(c\cdot(1-\alpha)-2) + P(E_2)\cdot 2 + \cdots + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)+1})\cdot(\frac{1}{2}\cdot c\cdot(1-\alpha)+1) + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)-1})\cdot(\frac{1}{2}\cdot c\cdot(1-\alpha)-1) + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)})\cdot\frac{1}{2}\cdot c\cdot(1-\alpha) > \frac{1}{2}(P(E_{c\cdot(1-\alpha)-1}) + P(E_1))\cdot(c\cdot(1-\alpha)) + \frac{1}{2}(P(E_{c\cdot(1-\alpha)-2}) + P(E_2))\cdot(c\cdot(1-\alpha)) + \cdots + \frac{1}{2}(P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)+1}) + P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)-1}))\cdot(c\cdot(1-\alpha)) + \frac{1}{2}\cdot P(E_{\frac{1}{2}\cdot c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha) = \frac{1}{2}\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)(c\cdot(1-\alpha)) = \frac{1}{2}\cdot(1 - P(E_{c\cdot(1-\alpha)}))(c\cdot(1-\alpha))$.

□

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 $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1) = (\beta-1)\cdot\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x = (\beta-1)\cdot(\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x + P(E_{c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha))$. Based on Theorem D.5.1, when $n \geq 2\cdot c\cdot(1-\alpha)\cdot\beta$, $\sum_{x=1}^{c\cdot(1-\alpha)-1} P(E_x)\cdot x > \frac{1}{2}\cdot(1-P(E_{c\cdot(1-\alpha)}))c\cdot(1-\alpha)$. Thus, $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1) > (\beta-1)\cdot(\frac{1}{2}\cdot(1-P(E_{c\cdot(1-\alpha)}))\cdot c\cdot(1-\alpha)+P(E_{c\cdot(1-\alpha)})\cdot c\cdot(1-\alpha))$, *i.e.*, $\sum_{x=1}^{c\cdot(1-\alpha)} P(E_x)\cdot x\cdot(\beta-1) > \frac{1}{2}\cdot(1+P(E_{c\cdot(1-\alpha)}))\cdot c\cdot(1-\alpha)\cdot(\beta-1) = \frac{1+\Pi_{i=1}^{c\cdot(1-\alpha)-1}\frac{n-i\beta}{n-i}}{2}\cdot c\cdot(1-\alpha)\cdot(\beta-1)$, *s.t.* $n \geq 2\cdot c\cdot(1-\alpha)\cdot\beta$.

APPENDIX E

SKIP DELTA-BASED VERSION CONTROL SYSTEMS

E.1 The Cost for Retrieving An Arbitrary Version in A Skip Delta-based Version Control System

Theorem E.1.1 shows that the cost of retrieving an arbitrary version in a skip delta-based version control system is bounded by $O(log(t))$, in which t is the number of versions stored in this version control system.

Theorem E.1.1. *In a skip delta-based version control system, the cost for retrieving an arbitrary version* t *is bounded by* $O(log(t))$.

Proof. (sketch) According to Figure 6.1(b), one can always re-compute version $\mathtt{F}_t$ by starting from the initial version $\mathtt{F}_0$, and applying all the corresponding skip deltas up to δ_t. Let l be the total number of skip deltas used to re-construct $\mathtt{F}_t$. Since the skip delta of an arbitrary version is the delta of this version against its skip version, l is thus equal to the total number of skip versions from $\mathtt{F}_0$ up to $\mathtt{F}_t$. According to the rule of determining a version's skip version in Section 6.3.1, l is actually the total number of bits with value "1" in t's binary format, which is at most $log(t)$. In other words, based on $\mathtt{F}_0$, one need to go through at most $log(t)$ skip deltas to re-compute $\mathtt{F}_t$. Thus, the cost for retrieving an arbitrary version t is bounded by $O(log(t))$. □

REFERENCES

[1] "Outsourcing," http://en.wikipedia.org/wiki/Outsourcing.

[2] "Amazon Web Services," http://aws.amazon.com/.

[3] "Windows Azure," http://www.windowsazure.com.

[4] "The Why of Cloud," http://www.gartner.com/DisplayDocument?doc_cd=226469&ref=g_noreg.

[5] "Amazon simple storage service," http://aws.amazon.com/en/s3/.

[6] "Amazon Gracier," http://aws.amazon.com/glacier/.

[7] 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.

[8] 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.*, vol. 14, June 2011.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.*, vol. 22, no. 5, May 2011.

[14] 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.

[15] 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.

[16] R. Curtmola, O. Khan, and R. Burns, "Robust remote data checking," in *Proc. of ACM Workshop On Storage Security And Survivability (StorageSS '08)*, 2008.

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

[18] A. G. Dimakis, B. Godfrey, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," in *Proc. of Annual IEEE Conference on Computer Communications (INFOCOM '07)*, 2007.

[19] 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*, vol. 56, Sept. 2010.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] B. Chen and R. Curtmola, "Auditable version control systems," in *Proc. of the 21th Annual Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[28] 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.

[29] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. of 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[30] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[31] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)*, 2000.

[32] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, pp. 190–201, November 2000.

[33] A. A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[34] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Workshop on Real-Life Cryptographic Protocols and Standardization*, 2010.

[35] S. Kamara, C. Papamanthou, and T. Roeder, "Cs2: A searchable cryptographic cloud storage system," in *Technical Report MSR-TR-2011-58*. Microsoft, 2011.

[36] D. Cash, A. Kupcu, and D. Wichs, "Dynamic proofs of retrievability via oblivious ram," in *Proc. of EUROCRYPT '13*, 2013.

[37] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," *Proc. of IEEE International Symposium on Network Computing and Applications (NCA '06)*, 2006.

[38] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *J. of the ACM*, vol. 36, no. 2, 1989.

[39] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," International Computer Science Institute, Tech. Rep. TR-95-048, August 1995.

[40] J. S. Plank, "Erasure codes for storage applications," Tutorial Slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html, San Francisco, CA, 2005.

[41] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on reed-solomon coding," 2003.

[42] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008.

[43] J. S. Plank, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications," University of Tennessee, Tech. Rep. CS-05-569, December 2005.

[44] "OpenSSL," http://www.openssl.org/.

[45] "Eclipse," http://archive.eclipse.org/arch/.

[46] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, May 1996.

[47] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Proc. of Annual International Cryptology Conference (CRYPTO '10)*, 2010.

[48] D. Boneh, D. Freeman, J. Katz, and B. Waters, "Signing a linear subspace: Signature schemes for network coding," in *Proc. of International Conference on Practice and Theory in Public Key Cryptography (PKC '09)*, 2009.

[49] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

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

[51] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained," in *Proc. of FAST'08*, 2008.

[52] B. Schroeder, S. Damouras, and P. Gill, "Understanding latent sector errors and how to protect against them," in *Proc. of FAST'10*, 2010.

[53] P. Maniatis, M. Roussopoulos, T. Giuli, D. Rosenthal, M. Baker, and Y. Muliadi, "The LOCKSS peer-to-peer digital preservation system," *ACM Transactions on Computer Systems*, vol. 23, no. 1, pp. 2–50, 2005.

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

[55] R. Gennaro, J. Katz, H. Krawczyk, and T. Rabin, "Secure network coding over the integers," in *Proc. of International Conference on Practice and Theory in Public Key Cryptography (PKC '10)*, 2010.

[56] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: a quantitative comparison," in *Proc. of International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.

[57] T. Ho, R. Koetter, M. Medard, D. R. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," in *Proc. of IEEE International Symposium on Information Theory (ISIT '03)*, 2003.

[58] T. Ho, M. Medard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Trans. Inform. Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

[59] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 265 – 279, 1981.

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

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

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology (CRYPTO '92)*, 1993.

[73] A. Fiat and A. Shamir, "How to prove yourself: practical solutions to identification and signature problems," in *Advances in Cryptology (CRYPTO86)*, 1986.

[74] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 2, pp. 299–327, 2005.

[75] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *Advances in Cryptology (Crypto '03)*, 2003.

[76] 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.

[77] C. Dwork, M. Naor, and H. Wee, "Pebbling and proofs of work," in *Advances in Cryptology (CRYPTO '05)*, 2005.

[78] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of space," http://eprint.iacr.org/2013/796.pdf.

[79] 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.

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

[81] J. M. Steele, *The Cauchy-Schwarz master class: An introduction to the art of mathematical inequalities*. Cambridge University Press, 2004.

[82] 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.

[83] W. F. Ehrsam, C. H. Meyer, J. L. Smith, and W. L. Tuchman, "Message verification and transmission error detection by block chaining," 1978, uS Patent 4,074,066.

[84] "Concurrent versions system," http://cvs.nongnu.org.

[85] "Apache subversion," http://subversion.apache.org/.

[86] "Git," http://git-scm.com.

[87] "Mercurial," http://mercurial.selenic.com.

[88] "Sourceforge," http://sourceforge.net.

[89] "Google code," http://code.google.com.

[90] "Github," https://github.com.

[91] "Dropbox," https://www.dropbox.com.

[92] "Bitcasa," https://www.bitcasa.com.

[93] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, p. 12, 2011.

[94] "Summary of the amazon ec2, amazon ebs, and amazon rds service event in the eu west region," http://aws.amazon.com/cn/message/2329B7/.

[95] "Summary of the aws service event in the us east region," http://aws.amazon.com/cn/message/67457/.

[96] "Gcc," http://gcc.gnu.org/.

[97] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Proc. of 6th IACR Theory of Cryptography Conference (TCC '09)*, 2009.

[98] G. Ateniese, S. Kamara, and J. Katz, "Proofs of storage from homomorphic identification protocols," in *Proc. of 15th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '09)*, 2009.

[99] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *Information Security*. Springer, 2001, pp. 379–393.

[100] "Code-sharing site github turns five and hits 3.5 million users, 6 million repositories," http://thenextweb.com/insider/2013/04/11/code-sharing-site-github-turns-five-and-hits-3-5-million-users-6-million-repositories/.

[101] "What is sourceforge.net [tm]?" http://sourceforge.net/apps/trac/sourceforge/wiki/What%20is%20SourceForge.net.

[102] "Summer of code 2012 ideas," https://github.com/trast/git/wiki/SoC-2012-Ideas.

[103] "Summer of code 2013 ideas," https://github.com/trast/git/wiki/SoC-2013-Ideas.

[104] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," internet RFC 2104, February 1997.

[105] "Apache portable runtime," http://apr.apache.org/.

[106] "Filezilla," https://filezilla-project.org/.

[107] "Wireshark," http://www.wireshark.org/.

[108] "Freebsd," http://www.freebsd.org/.

[109] "The apache software foundation," http://www.apache.org/.

[110] "Wget," http://www.gnu.org/software/wget/.