EFFICIENT PROACTIVE SECURITY FOR SENSITIVE DATA STORAGE

A Thesis
Presented to
The Academic Faculty

by

Arun Subbiah

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2007

EFFICIENT PROACTIVE SECURITY FOR SENSITIVE DATA STORAGE

Approved by:

Professor Douglas M. Blough,
Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Douglas M. Blough, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Faramarz Fekri
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Date Approved: 23 August 2007

ACKNOWLEDGEMENTS

I thank my advisor Prof. Douglas Blough for guidance and support throughout graduate school, and for the opportunity to pursue a PhD under his advisement. I thank Prof. Faramarz Fekri, Prof. Sudhakar Yalamanchili, Prof. Linda Wills, and Prof. Mustaque Ahamad for serving on my PhD committee and for their valuable comments. I had the opportunity to work with Prof. Mustaque Ahamad in the Agile Store project and it was a pleasure.

It was a pleasure to work with Lei Kong, Deepak Manohar and Michael Sun on the Agile Store project. They were very willing to discuss research ideas.

On a philosophical note, achieving the PhD can be attributed to not only the individual but also to that individual's environment, which is a cumulation of factors that date at least as far back in time as the individual's memory can recall. I consider myself blessed to have that conducive environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Fault tolerant and secure distributed data storage systems typically require that only up to a threshold of storage nodes can ever be compromised or fail. In proactively-secure systems, this requirement is modified to hold only in a time interval (also called epoch), resulting in increased security. An attacker or adversary could compromise distinct sets of nodes in any two time intervals. This attack model is also called the mobile adversary model. Proactively-secure systems require all nodes to "refresh" themselves periodically to a clean state to maintain the availability, integrity, and confidentiality properties of the data storage service.

This dissertation investigates the design of a proactively-secure distributed data storage system. Data can be stored at storage servers using encoding schemes called secret sharing, or encryption-with-replication. The primary challenge is that the protocols that the servers run periodically to maintain integrity and confidentiality must scale with large amounts of stored data. Determining how much data can be proactively-secured in practical settings is an important objective of this dissertation.

The protocol for maintaining the confidentiality of stored data is developed in the context of data storage using secret sharing. We propose a new technique called the GridSharing framework that uses a combination of XOR secret sharing and replication for storing data efficiently. We experimentally show that the algorithm can secure

several hundred gigabytes of data.

We give distributed protocols run periodically by the servers for maintaining the integrity of replicated data under the mobile adversary model. This protocol is integrated into a document repository to make it proactively-secure. The proactively-secure document repository is implemented and evaluated on the Emulab cluster (http://www.emulab.net). The experimental evaluation shows that several hundred gigabytes of data can be proactively secured.

This dissertation also includes work on fault and intrusion detection - a necessary component in any secure system. We give a novel Byzantine-fault detection algorithm for quorum systems, and experimentally evaluate its performance using simulations and by deploying it in the AgileFS distributed file system.

CHAPTER I

INTRODUCTION

The increasing dependence on the Internet and network-based services has brought with it the critical requirement that such services are fault-tolerant and secure. Today's critical computer systems must implement layers of defense against attacks and faults. The three layers of defense against faults and attacks (Figure 1) are - fault and intrusion prevention, fault and intrusion detection, and fault and intrusion tolerance. Examples of fault and intrusion prevention are firewalls, authentication, access control, and voltage spike suppressors. Examples of fault and intrusion detection are TripWire [4] and CRC (Cyclic Redundancy Check). Examples of fault and intrusion tolerance are RAID, filesystem backups, and secret sharing.

This dissertation focusses on the fault and intrusion detection and tolerance layers in the context of a distributed *storage* system. Storage is a critical and a fundamental component in any system. Providing storage services over the network is widespread, an example being the NFS file service. Other examples include web-based email and employee records, which are all stored at remote servers and can be accessed by authorized clients. In these examples, the interface exported to the user and the way in which data is manipulated may differ, but the common property is that the data is ultimately stored at remote servers, and the data is sensitive in nature.

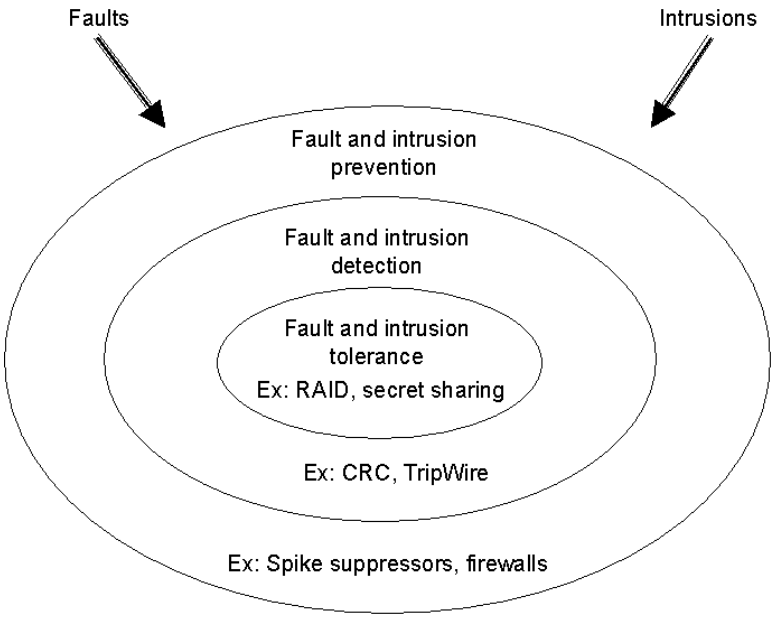

Figure 1: The three layers of defense against faults and intrusions.

The demand for a variety of always-available services over the network and the commensurate increase in the computing resources available to users and attackers has broadened the scope of sensitive information from obviously sensitive material, such as cryptographic keys and passwords, to generic data that needs to be protected from select people while being irrelevant to others. The *sensitivity* of data has now become a relative term, in terms of time as well as human perception.

This dissertation investigates the design of a highly reliable and secure data storage service that stores sensitive data. The storage service is provided by a set of *storage servers*. A schematic overview of a typical data storage service is shown in Figure 2. The security of a data storage service is defined as providing the following three properties in the presence of server compromises and failures:

1. Availability: The ability to always provide the data storage service.

2. Integrity: The stored data is not lost or corrupted, and the read-write semantics

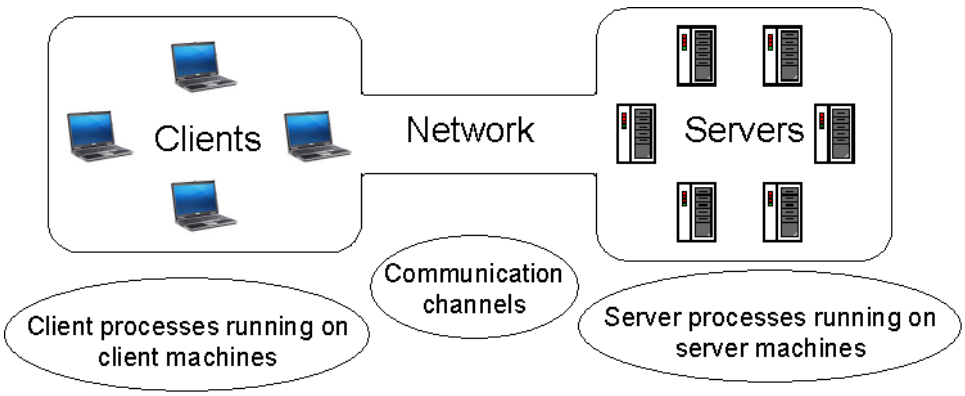

Figure 2: Schematic overview of a data storage service.

continue to hold.

3. Confidentiality: The stored data is not revealed to the adversary.

The computer security community has investigated securing data storage services using a variety of assumptions on the adversary compromising the storage servers. These assumptions revolve around either the adversary's *intent*, or the adversary's *limitations*. The adversary's intent is used to classify an adversary as an active adversary or a passive adversary. A passive adversary compromises *processes* or *communication channels* to learn their stored data, internal states and messages, thus affecting the confidentiality of the system. An active adversary is not only intent on breaking the confidentiality of the system, but also the integrity of the system. An active adversary thus corrupts stored data, internal states and messages. Examples of an adversary's *limitations* are: being computationally bounded, being memory bounded, and the rate at which an adversary can compromise processes and communication channels.

The limitation on the rate at which an adversary can compromise processes is known as the mobile adversary model. In this model, real time is divided into *epochs*,

and it assumed that only some processes are in the compromised state in an epoch. A system that is resilient to the mobile adversary is said to be *proactively secure*.

The dependable computing community has meanwhile investigated this area using a variety of fault models. One of the common fault models used is the *hybrid* fault model [61], where the processes in a system can exhibit Byzantine and crash faults. We combine the hybrid fault model with the passive and active adversary models to give the BLOC fault model, described in Section 1.1. The mobile adversary model, in the context of the BLOC fault model, and the concept of proactive security are described in Section 1.2. This dissertation investigates data storage services resilient to the BLOC fault model along with the mobile adversary model[1]. Section 1.3 gives the contributions made in this research, and Section 1.4 gives an organizational overview of this dissertation.

1.1 The BLOC Fault Model

The *passive* and *active* adversary models used in computer security and the Byzantine and crash fault models for processes (also known as the *hybrid* fault model [61]) used in dependable computing are combined to give the *BLOC* fault model, where any process can undergo one of the following three faults:

- **Crash:** A process is said to be *crashed* if it stops performing all computations and neither sends nor receives messages on the network.

- **Byzantine:** A Byzantine-faulty process can deviate arbitrarily from its specified protocol behavior. A Byzantine faulty process can also reveal the data stored locally and its internal state to an adversary. This fault is due to the compromise by an *active* adversary.

[1]The adversary is also assumed to be computationally bounded.

- **Leakage-only:** A process is said to exhibit a leakage-only fault if it can reveal its stored data and state to an adversary, but executes its specified protocol faithfully. This fault is due to the compromise by a *passive* adversary.

We use the threshold fault assumption for each of the three types of faults. We assume that, in a system containing n processes, not more than c processes can crash, not more than b processes can be Byzantine-faulty, and not more than l processes can exhibit leakage-only faults. In this dissertation, the servers and the clients are the *processes*, and the threshold fault model is used only for the servers; an unbounded number of clients can exhibit any of the three faults.

The proposed fault model allows for direct reasoning about the availability, integrity, and confidentiality properties of the storage service. In availability attacks, such as Denial-of-Service attacks, the resources available for legitimate use of the service are constrained by, for example, exhausting network bandwidth or by increasing server loads. Crash faults are representative of a more severe attack, where a server stops execution completely and permanently. A storage service that can tolerate a high number of crash faults is also a highly-available storage service, and will be able to tolerate Denial-of-Service attacks to a greater degree. Integrity attacks, in the context of a storage service, can consist of either compromising servers and altering their behavior, or compromising servers and arbitrarily corrupting their stored data. Such attacks are represented by Byzantine faults. The confidentiality of the stored data can be compromised if an adversary is able to compromise sufficient servers and learn their stored data. These are modeled by the Byzantine and leakage-only faults.

1.2 Proactive Security and the Mobile Adversary Model

The mobile adversary model was introduced in [51]. In this model, real time is divided into *epochs*, and no more than a threshold servers can be in the compromised state

in an epoch. In other words, the adversary "moves" from one server to another, with the restriction that only up to a threshold servers are "visited" by the adversary in an epoch.

In the context of the BLOC fault model, since server compromises are modeled using Byzantine and leakage-only faults, only up to a threshold b of servers can be Byzantine-faulty and a threshold l of servers can be leakage-only faulty in an epoch. Thus, after a sufficient many epochs elapse, all the servers may have experienced a Byzantine fault or a leakage-only fault in some epoch(s).

A system that is secure against the mobile adversary is said to be *proactively secure*. Such a system must run some procedures periodically (such as once every epoch) so that the availability, integrity, and the confidentiality properties of the storage service are maintained.

1.3 Contributions

The goal of this research is to investigate the design of a proactively-secure data store. Contributions made as part of this research are as follows:

- For storing data, we consider, in addition to the standard encrypt-and-replicate storage model, data storage using perfect secret sharing schemes. We show that standard secret sharing schemes have high computation overheads and are impractical for storing large amounts of data. A new technique called the GridSharing framework is proposed [60] that uses a combination of XOR secret sharing and replication for storing data efficiently. The number of rows in the GridSharing framework is a configurable parameter that can be varied to achieve a tradeoff in performance metrics.

- We give a share renewal algorithm in the context of the GridSharing framework

for maintaining the confidentiality of the stored data under the mobile adversary model. We experimentally show that the algorithm can secure several hundred GBs of data.

- We give distributed protocols run periodically by the servers for maintaining the integrity of replicated data under the mobile adversary model. We experimentally show that these protocols scale to several 100 GBs of stored data.

- We design a proactively-secure document repository, where users can upload a new version of a document, download the latest version of a document, and delete documents. The read-write protocols are specified. The protocol run periodically for maintaining the integrity under the mobile adversary model is integrated into the prototype, and concurrency and timing issues at epoch boundaries are addressed. The result is an integrated system that maintains correctness in the presence of concurrent executions of the read-write protocols and the protocol for maintaining integrity in the mobile adversary model.

- The proactively-secure document repository is implemented and evaluated on the Emulab cluster (http://www.emulab.net). The experimental evaluation shows that several 100 GBs of data can be proactively-secured.

- A necessary component in any secure system is fault and intrusion detection. We give a novel Byzantine-fault detection algorithm for quorum systems [39], and experimentally evaluate it using simulations and by deploying it [38] in the AgileFS distributed file system. Concurrent reads and writes are a source for false alarms in Byzantine fault detection algorithms. We show that the proposed fault detection algorithm does not produce incorrect diagnosis even when the read-write concurrency rate is as high as 32%. In addition, an interesting

property of the fault detection algorithm is that a Byzantine-faulty server has to mimic closely a fault-free server to avoid being detected.

1.4 Dissertation Organization

This dissertation is organized as follows:

Chapter 2 gives an overview of other works on secure distributed storage services and systems.

Chapter 3 describes the architecture of the reference data storage service, and outlines the protocols that must be executed periodically for the service to be proactively secure.

The confidentiality of the stored data can be achieved using either encryption or secret sharing schemes. Both these techniques are considered in this research.

Chapter 4 describes a novel secret sharing algorithm, called the *GridSharing Framework*, that uses a combination of XOR secret sharing and replication for computational efficiency. The computational efficiency is important during reads and writes and during a procedure called "share renewal", in which the shares of the encoded data are changed using distributed protocols such that the encoded data still remains the same.

Chapter 5 describes the share renewal protocol for the GridSharing framework. This share renewal protocol is run periodically in the GridSharing framework to maintain the confidentiality of the stored data in the presence of the mobile adversary.

Chapter 6 describes Protocol PIVR (Periodic Integrity Verification and Repair) executed by the servers to maintain the integrity of replicated data in the presence of a mobile adversary.

Chapter 7 describes a fault detection algorithm run by the servers to detect Byzantine faults. Fault and intrusion detection mechanisms are required even in the mobile

adversary model; without such mechanisms, the adversary will *spread* to all the servers as opposed to *moving* from one server to another.

Chapter 8 describes the implementation and experimental evaluation of a proactively-secure data store where encrypted data is stored using replication.

Chapter 9 wraps up the dissertation with summary points and some directions for future research.

CHAPTER II

RELATED WORK

One of the earliest distributed data storage systems is RAID [52]. Some RAID configurations can be used to protect the integrity and availability of the stored data in the event of hardware failures.

The field evolved into distributed data storage systems that use replication or some type of coding scheme such as erasure codes to store data reliably at a set of *storage servers*. Examples of such systems are [6, 7, 8, 11, 22, 32, 37, 38, 40, 45, 46, 25, 62, 27]. Data confidentiality, an important security guarantee that must be provided when the data is sensitive, is provided by storing the data in encrypted form. The encryption keys are assumed to be managed securely by the user(s), or by storing them using *perfect* secret sharing schemes. In *perfect* secret sharing schemes, data is encoded into *shares* such that the knowledge of some number of shares to an attacker gives no information on the encoded data. If the shares are distributed amongst the storage servers, then the confidentiality of the stored data is provided by assuming that only up to a certain threshold of servers can be compromised.

Instead of encrypting the data and storing the encryption keys using perfect secret sharing, the data can be stored directly using perfect secret sharing. Examples of works where data is stored using such schemes are [42] and [65]. This dissertation

considers data stored using perfect secret sharing as well as encrypt-and-replicate.

An overwhelming majority of existing works assume a a bound on the number of servers that can become faulty or compromised over the entire lifetime of the system. This bound is called the *fault threshold*. In the context of providing long-term security, this fault-threshold assumption is unreasonable. The mobile adversary model, introduced in [51], addresses this problem. In this model, it is recognized that an adversary can be powerful enough to compromise all the data servers given enough time. The problem is made tractable by assuming that no more than a threshold number of servers can be compromised or become faulty in a given time window. A system that is secure against the mobile adversary is said to be *proactively-secure*. Examples of works on proactively-secure services are agreement [56], secret sharing [51, 35, 58, 16, 68, 69], signatures [34], RSA [30, 55], pseudo-randomness [19, 24], and clock synchronization [13]. A survey of works on proactive security can be found in [17]

Proactive secret sharing was also introduced in [51]. In proactive secret sharing, data is encoded using perfect secret sharing schemes. Periodically, the shares are changed using a distributed protocol, and this process is called share renewal. Thus, an adversary must compromise enough servers in a single time window to retrieve the requisite number of shares to recover the encoded data, which is clearly much harder for the adversary to do compared to the case where the entire system lifetime is available for the compromise of the servers.

In [35], a robust proactive secret sharing scheme is given. Here, the model to counter the mobile adversary was refined to specify that all the servers must be rebooted periodically. The reboot corrects any faulty servers. Periodic reboot is necessary because the underlying fault and intrusion detection mechanisms may not be reliable. The servers engage in the share renewal protocol after their reboot. In

addition, [35] recognized that an adversary may modify the shares stored at compromised servers thus affecting the integrity of the stored data. So a share integrity and share recovery procedure must also be done periodically. [35] gives a *robust* proactive secret sharing scheme that takes long-term data integrity and confidentiality into consideration.

We investigate a proactively secure data storage service where the data is stored using the encrypt-and-replicate storage model as well as perfect secret sharing. For protocol designs, barring [16, 21, 67], all works on proactive security assume the network to be a broadcast channel with zero or negligible message delays.In [16, 21, 67], however, an asynchronous system model is assumed. The protocols given in [16, 21, 67] incur high computation and communication overheads and are unsuitable for managing large amounts of data. The protocols for proactive security that we develop in this disseration assume a synchronous system model with point-to-point network links which have non-zero message delays.

We differ from prior works on proactive security by addressing the problem of providing proactive security efficiently for *large* amounts of data. A synchronous system model is used for keeping data proactively secure. This research has its roots in the Agile Store project [1] at Georgia Tech. The Byzantine-fault detection algorithm for quorum systems that is part of this research was used in reconfigurable Byzantine quorum systems [39] for the Agile Store. The Agile Store project primarily addresses the problem of estimating the fault threshold and keeping it low, which would help improve the efficiency of the system. In the proactive model, the fault threshold is maintained at a small value using such Byzantine-fault detection techniques and by performing certain procedures periodically, such as integrity verification and share renewal.

CHAPTER III

DATA STORE ARCHITECTURE

This chapter describes the architecture of the data store considered in this dissertation. The architecture is shown in Figure 3.

3.1 System Model

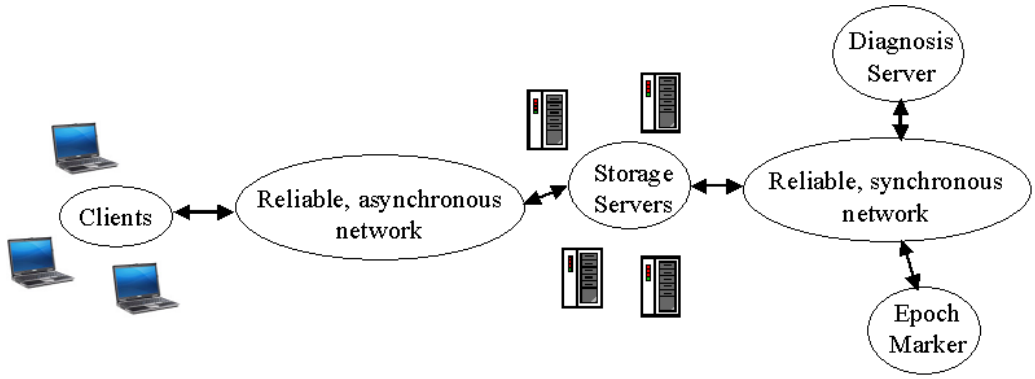

Figure 3: System architecture

The system architecture, shown in Figure 3, consists of the following entities:

1. Storage Servers: which provide the data storage service.

2. Diagnosis Server: which collects fault detection information from the storage servers and determines if a storage server is Byzantine-faulty.

3. Epoch Marker: which divides real time into time periods and notifies the storage servers and the diagnosis server the start of a new time period.

4. Clients: which read and write data to the storage servers.

The Epoch Marker divides real time into epochs. It notifies the start of a new epoch to the storage servers and the diagnosis server. Upon learning the start of a new epoch, the servers perform certain tasks to achieve proactive security.

This research includes work on server fault detection, where the storage servers diagnose each other as being *Byzantine-faulty* or not by observing server responses during read operations. The storage servers communicate their diagnosis results to the diagnosis server, which uses voting to ultimately determine whether a storage server is Byzantine-faulty.

3.1.1 Failure Assumptions

The BLOC fault model, given in Section 1.1, is used to describe the failures that may occur at the servers. The diagnosis server is assumed to be always fault-free, while the epoch marker can only be leakage-only faulty. For the storage servers, the BLOC fault model together with the mobile adversary model is used. It is assumed that, in an epoch (the start and end times of which are determined by the epoch marker), not more than b storage servers can be Byzantine-faulty and not more than l storage servers can be leakage-only faulty. Not more than c storage servers can crash throughout the lifetime of the system.

A fault-free server could have its stored data in a corrupted state or could have revealed its stored data to the adversary if it was Byzantine-faulty or Leakage-Only faulty in an earlier epoch.

There is no restriction on the number of clients that can be Byzantine-faulty.

3.1.2 Communication Model

The servers communicate with each other over a point-to-point fully-connected reliable and synchronous network. In a reliable network, if Process i sends a message to Process j, then Process j will receive the message. In a synchronous network, all messages are guaranteed to be delivered to the intended recipients in bounded time. It is assumed that any computations done at the servers also finish in bounded time.

The clients read and write data to the storage servers by communicating over a reliable and asynchronous network. In an asynchronous network, there is no upper bound on the message delays. Computations done at the clients and the servers can take an arbitrarily long time.

All messages are assumed to be authenticated, confidential, and tamper-proof. These are achieved using cryptographic techniques. The clients cannot communicate with the diagnosis server and the epoch marker.

3.2 Data Storage Models

We consider two data storage models in this dissertation - secret sharing, and encrypt-and-replicate. These two storage models are described next.

3.2.1 Secret Sharing

In the *secret sharing* data storage model, data is encoded into *shares* such that any threshold of these shares can be later used to recover the encoded data. Each share is stored at one or more servers.

The stored data remains confidential as long as sufficient many servers do not become Byzantine or leakage-only faulty and reveal their shares to the adversary. There are no encryption or decryption keys involved.

The adversary can arbitrarily corrupt the shares stored at a Byzantine-faulty

server. To enable clients to detect corrupted shares, verifiable secret sharing schemes are typically used. In such schemes, some public information on all the shares is computed at the time of share generation and is stored at all the storage servers. When reading data, clients compare the shares against this public information to determine if the share is corrupted. Verifiable secret sharing schemes are computationally expensive to be used on large amounts of data. The GridSharing framework, described in Chapter 4, addresses this problem and gives a practical method for storing large volumes of data using secret sharing.

3.2.2 Encrypt and Replicate

In this model, the data is encrypted by the user, and the encrypted data is stored at all the storage servers. If some server becomes Byzantine or leakage-only faulty, then the encrypted data can be revealed to the adversary. The confidentiality of the encrypted data relies on the secure maintenance of the decryption keys. The decryption keys can be managed privately by the users(s), or stored using secret sharing at the storage servers, or stored using the CODEX system [47], wherein the distributed data storage service has a public key and the decryption key is encrypted using this public key and stored at all the servers.

3.3 Achieving Proactive Security

The data storage service is said to be *proactively secure* if it is resilient to the mobile adversary. We consider the mobile adversary model only for the storage servers. In an epoch, some threshold number of servers can become Byzantine or leakage-only faulty; it is not required that the same servers are Byzantine or leakage-only faulty in different epochs. Maintaining the availability, integrity, and confidentiality properties of the stored data require certain tasks to be executed periodically by the servers. An

Table 1: Possible sharings generated during share renewal in every epoch.

Epoch	Encoded bit s1 $\oplus$ s2 $\oplus$ s3	Server S_1 Share s1	Server S_2 Share s2	Server S_3 Share s3
1	1	0	0	1
2	1	1	0	0
3	1	1	1	1
4	1	1	0	0
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

overview of these tasks are described in this section.

3.3.1 Periodic Share Renewal

Share renewal is relevant when the data is stored using the secret sharing data storage model. During share renewal, the servers execute a distribute protocol to generate new shares of the encoded data without learning the encoded data in the process. Performing share renewal periodically will maintain the confidentiality of the encoded data under the mobile adversary model.

Consider the following example: There are three storage servers S_1, S_2, and S_3. An adversary can compromise a maximum of one server in an epoch and thus create the leakage-only fault. Consider XOR secret sharing. Consider one bit of the encoded data. For this bit, three shares, each also one-bit long, are generated such that the XOR of all three shares gives the encoded bit. Table 1 gives a possible set of sharings of the encoded bit generated during the share renewal process run in every epoch.

Thus, if not more than one server is leakage-only faulty in every epoch, an adversary will never be able to obtain all three shares *of the same sharing*. Performing share renewal periodically will thus maintain the confidentiality of the stored data under the mobile adversary model. Chapter 5 describes the share renewal protocol

for the GridSharing framework of storing data using secret sharing.

3.3.2 Periodic Integrity Verification and Repair

The Periodic Integrity Verification and Repair (PIVR) protocol is run by the servers periodically to maintain the integrity of the stored data under the mobile adversary model. In this protocol, a server checks if the checksums of its stored data match with other servers. The server can thus detect if its stored data is corrupted, and can then repair it by reading the correct data off other servers.

This procedure is relevant in any model where the data, or at least the checksums or some verifying information of the data, is stored using replication. In the encrypt-and-replicate storage model, the encrypted data is replicated at all the servers. In the GridSharing framework, each share is managed using replication. Most secret sharing or coding schemes store checksums or some public information on the encodings using replication. Thus, Protocol PIVR has a broad application. Chapter 6 describes and analyzes this protocol.

3.3.3 Periodic Reboot and Key Establishment

The assumption that the Byzantine and leakage-only faults move from one server to another, as opposed to spreading to all servers, can be realized only by having the servers restore themselves to a clean state periodically. This requires the servers to securely reboot from a read-only storage medium every epoch.

Compromise of a server also gives away the cryptographic keys used for communication between the servers and with the clients. The adversary may have "left" the server, but it can listen to and modify the messages in the network. To prevent this, after the reboot, the servers must re-establish new cryptographic keys for securing network communication. Techniques to achieve this are given in [18, 12].

CHAPTER IV

THE GRIDSHARING FRAMEWORK

As mentioned in Section 3.2, this dissertation considers two storage models - encrypt-and-replicate and secret sharing. This chapter gives a practical secret-sharing technique, called the GridSharing framework, for storing large amounts of data.

4.1 Introduction

Data confidentiality, an important security guarantee that must be provided for sensitive data, is provided either by storing the data in encrypted form, or storing the data using secret sharing, or by a combination of both. This chapter considers the problem of storing data using secret sharing.

Secret sharing schemes can be classified into *perfect* and *imperfect* schemes. Perfect secret-sharing schemes encode data into *shares* such that only certain valid combinations of shares can be used to reconstruct the encoded data, while invalid combinations of shares give no information about the encoded data. By storing these shares at different servers, the encoded data is kept confidential as long as sufficiently many servers are not compromised. Examples of perfect secret-sharing schemes are Shamir's scheme [57] and Blakley's scheme [15]. In *imperfect* secret sharing schemes, invalid combinations of shares can give some partial information about the encoded data. Examples of imperfect secret-sharing schemes are erasure codes and Rabin's

IDA algorithm [54]. We consider only perfect secret-sharing schemes in this dissertation.

Confidentiality is achieved without any encryption, thus avoiding the need for the storage and management of cryptographic keys. Perfect secret-sharing schemes have the additional property that the shares can be changed, or *renewed*, distributively such that the encoded data still remains the same. This process of share renewal, when performed often, can provide strong data confidentiality. The security of such a scheme relies on the inability of an adversary to compromise a sufficient number of servers in the time between two consecutive share renewals.

Unlike private-key encryption schemes, however, most perfect secret-sharing schemes are computationally expensive. Verifiable secret-sharing schemes [23] are typically used with perfect secret-sharing schemes to detect corrupted shares that may be returned by compromised servers, and also to detect incorrect secret sharing during writes by malicious clients. Such techniques further increase the computation time during the encoding and decoding of data. Perfect secret-sharing schemes have thus found little use in storing data because of their high computation overhead.

We solve these problems by 1) using XOR secret sharing for fast computations, and 2) using replication-based schemes to detect corrupted shares that may be returned by Byzantine-faulty servers. This combination of secret sharing and replication manifests itself as an architectural framework where servers are arranged in the form of a rectangle or a grid. The proposed architectural framework, which we call *the GridSharing framework*, has the useful property that its dimensions can be varied to trade off several performance metrics. The properties and performance of the GridSharing framework are presented in this chapter.

4.2 Related Work

In secret sharing, data is encoded into several *shares* such that only certain combinations of shares can be used to reconstruct the encoded data. Most works use imperfect secret-sharing schemes, such as erasure codes (e.g., Rabin's IDA [54] algorithm), where the knowledge of fewer than the threshold number of shares can reveal some information about the encoded data. Such coding algorithms are thus not information-theoretic secure, but allow savings in storage space. Given enough time, an adversary may compromise enough servers to learn the encoded data. Thus, to provide long-term confidentiality, the secret-sharing scheme should allow share renewal, where the shares are changed in a distributed fashion such that the encoded secret is not recovered in the process and is unchanged. To our knowledge, no distributed share renewal scheme for imperfect secret sharing has been developed to date. We instead use perfect secret-sharing schemes (example, Shamir's scheme [57]), which allow distributed share renewal. Perfect secret-sharing schemes are also information-theoretic secure, meaning the leakage of an insufficient number of shares to an adversary does not reveal any information about the encoded data.

When data is stored using secret sharing, it must be possible for a client to identify corrupted shares during reads. Verifiable secret-sharing schemes [23] can be used with perfect secret-sharing for this purpose and also to check if the secret sharing was performed correctly during writes. Verifiable secret-sharing schemes also allow share renewal. However, such schemes are computationally expensive. Section 4.4 describes in detail how we avoid using verifiable secret-sharing schemes, thereby drastically reducing the computation overhead. Another approach to detect corrupted shares during reads is to store the hash of the shares in a hash vector at all the servers. To our knowledge, no algorithm has been developed for updating the hash vector

distributively after share renewal.

Several works have combined replication-based mechanisms and perfect secret sharing [33, 50, 42]. In [33], data is encrypted using a key, and both are stored at the storage servers. The data is stored in replicated form in a quorum, while the key is stored using secret sharing. In [50], quorum systems and secret sharing are used to build an authorization service. Quorum properties are used to ensure that sufficient servers agree to authorize a request, but the shares are not replicated at the servers. The paper addresses malicious users and does not consider compromised servers. The shares are never directly read and written. Thus, [33, 50] consider using perfect secret sharing for some special types of data and not for generic data. Performance during reads and writes is not addressed. In [42], perfect secret sharing is used for generic data, while [64] uses perfect secret sharing for archival data. Both these works do not address the problem of high computation overhead.

In [26], a technique called fragmentation-redundancy-scattering is used. The security of this scheme relies mainly on the secure maintenance of the encryption key and the fragmentation key. We instead store data directly using perfect secret-sharing schemes.

XOR secret sharing has been considered in [41]. The authors show how different capabilities such as share renewal and share recovery can be implemented with XOR secret sharing. For this, the existence of a trusted device, called the *Accumulator*, is assumed. They also assume that no server returns erroneous responses during secret sharing and secret recovery. The performance benefits associated with the use of XOR secret sharing are not discussed.

In [66], secret sharing is used to build survivable information storage systems. The tradeoffs possible when using p-m-n threshold schemes are outlined. The description is in terms of how the choice of p, m, and n affect performance. In this chapter,

we not only explore the tradeoff space in detail, but also address the performance overheads involved in such schemes. However, we consider only perfect secret-sharing schemes (which are a special case of p-m-n schemes), since distributed share renewal algorithms (e.g., [35]) have been developed only for these schemes.

4.3 Background

4.3.1 Secret Sharing Schemes

Secret-sharing schemes are techniques in which a *secret* is encoded into several fragments, called *shares*, such that certain combinations of shares can together reveal the encoded secret. In *perfect* secret-sharing schemes, invalid combinations of shares give no information about the encoded secret. Thus, perfect secret-sharing schemes are information-theoretic secure. Perfect secret-sharing schemes also allow share renewal, which is the process of distributively changing the shares such that the encoded secret is the same. Frequent share renewal can provide strong data confidentiality.

In perfect *threshold* secret-sharing schemes, a secret is encoded into q shares such that any k out of the q shares can be used to recover the encoded secret, while any $(k-1)$ shares give no information about the encoded secret. Such schemes are also called (k, q)-threshold schemes. Shamir's scheme [57] is an example of a (k, q)-threshold perfect secret-sharing scheme, where $k \leq q$.

In the next subsection, we describe Ito, Saito, and Nishizeki's share assignment scheme [36], which realizes any access structure using a (q, q)-threshold secret-sharing scheme.

4.3.2 Ito, Saito, and Nishizeki's Share Assignment Scheme

We describe Ito, Saito, and Nishizeki's share assignment scheme [36] for a threshold access structure. Consider a set of r participants $\{P_1, P_2, ..., P_r\}$ such that any $(m+1)$

participants can pool their shares to recover the encoded secret. For a secret sharing scheme realizing this access structure, first list the set B consisting of all possible combinations of m participants. Thus, $B = \{B_1, B_2, ..., B_{\mathrm{q}}\}$, where $q = \binom{r}{m}$.

Next, encode the secret using a (q, q)-threshold secret sharing scheme, where $q = \binom{r}{m}$. Let the shares thus generated be denoted by $s = \{s_1, s_2, ..., s_{\mathrm{q}}\}$, where $q = \binom{r}{m}$. The set of shares assigned to participant P_{i} is given by the function $g(i) = \{s_{\mathrm{j}}, P_{\mathrm{i}} \notin B_{\mathrm{j}}, 1 \leq j \leq q\}$. Thus, each participant receives $\binom{r-1}{m}$ shares, and each share is stored at $(r - m)$ participants.

For example, consider a set of four participants such that at least three participants must pool their shares to find the encoded secret. Then, $r = 4$, $m = 2$, and the set $B = \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_3), (P_2, P_4), (P_3, P_4)\}$. Next, generate six shares of the secret such that all six of them are needed to decode the secret. Denote the six shares by $\{s_1, s_2, s_3, s_4, s_5, s_6\}$.

From the share assignment function g,

Participant P_1 gets shares (s_4, s_5, s_6),

Participant P_2 gets shares (s_2, s_3, s_6),

Participant P_3 gets shares (s_1, s_3, s_5),

Participant P_4 gets shares (s_1, s_2, s_4).

Thus, any two participants can pool their shares to find out only five of the six shares. Without the knowledge of the sixth share, the encoded secret cannot be learnt. Any three participants can pool their shares to find out all six shares needed to recover the encoded secret.

4.4 Computation Overhead of Perfect Secret Sharing Schemes

In this section, we show the high computation overhead of some well-known secret sharing schemes, which is the main reason why such schemes are not widely used for distributed data storage. We contrast the computation overhead with that of the Rijndael (AES) symmetric-key encryption algorithm to illustrate this point. We then show that XOR secret-sharing combined with replication-and-voting mechanisms has a computational overhead similar to that of the Rijndael algorithm. All performance measurements reported were done on an Intel Pentium4 3GHz processor with 256 MB RAM running Linux 2.6.9. The MIRACL [3] library was used to implement all the cryptographic algorithms. In Section 4.7, we also compare the communication overhead of the techniques developed against encryption and secret sharing.

Table 2: Computation times for Shamir's scheme (8 KB block)

Prime Length	$(k, q) = (3, 5)$		$(k, q) = (6, 10)$	
	Sharing	Recovery	Sharing	Recovery
160 bits	4.956 ms	826 μs	14.87 ms	1.446 ms
512 bits	6.192 ms	1.290 ms	20.00 ms	2.064 ms
1024 bits	10.53 ms	2.145 ms	34.65 ms	3.575 ms

Shamir's scheme [57] is an example of a (k, q)-threshold perfect secret-sharing scheme, where $k \leq q$. Table 2 lists the time taken to compute shares (sharing), and the time taken to compute the secret given enough shares (recovery), for an 8 KB block of data using Shamir's scheme, for a selection of (k, q) values. Secret sharing and recovery are done during writes and reads, respectively, and their overheads are therefore important. For Shamir's scheme, since the computations are done modulo a prime p, the size of this modulus is also a factor in the throughput measurements.

There are two attacks possible when data is stored using secret sharing techniques. One attack is by a faulty client that generates inconsistent shares during writes, i.e.,

different subsets of k shares out of the q shares will decode to different values. The other attack is when a faulty server returns incorrect or arbitrary shares during reads. Such attacks can be detected using verifiable secret-sharing schemes [23]. In such schemes, some common data (called *witnesses*) for all the shares is computed by a client during writes and sent to all the servers. Before storing the shares and the witnesses, the servers check the shares received against the witnesses and arrive at a consensus on the consistency of the shares. During reads, a client will first determine the witnesses and check the validity of each share with the witnesses before proceeding to decode the sensitive data. Verifiable secret-sharing schemes significantly increase the computation overhead during the secret sharing (encoding) and secret recovery (decoding) processes. A widely used method for verifiable secret sharing is Feldman's scheme [28]. Table 3 gives the computation times during secret sharing and secret recovery of an 8 KB block of data when Feldman's scheme is used with Shamir's scheme.

Table 3: Computation times for 8 KB block using Shamir's with Feldman's scheme (Feldman's prime length = 1025 bits).

Prime Length	$(k, q) = (3, 5)$		$(k, q) = (6, 10)$	
	Sharing	Recovery	Sharing	Recovery
160 bits	2.461 s	2.616 s	4.956 s	7.228 s
512 bits	1.037 s	1.097 s	2.090 s	2.795 s
1024 bits	728 ms	747.5 ms	1.464 s	1.809 s

For comparison purposes, the throughputs of the AES Rijndael symmetric-key encryption algorithm are given in Table 4.

From Tables 2–4, it is clear that the computation times of Shamir's scheme and Feldman's scheme are far higher than those of symmetric-key encryption and, in fact, this performance is well below what is acceptable for modern data storage systems. The secret recovery computation times for verifiable secret sharing are at least 3000

Table 4: Computation time for AES (CBC mode, 8 KB block).

Key length	Encryption	Decryption
16 bytes	205 μs	205 μs
24 bytes	230 μs	241 μs
32 bytes	282 μs	271 μs

times slower than the Rijndael decryption times. The above analyses also indicate, in part, why perfect secret-sharing techniques have not been adopted for generic data to date. We reduce the computation overhead by using the following two mechanisms:

Mechanism 1: Use a (q, q) perfect secret sharing scheme: When $k = q$, i.e., all the shares are needed to recover the secret, then "inconsistent" secret sharing is not possible. That is, there is no question of different subsets of k shares out of q shares decoding to different values because there is only one such subset, since $k = q$. Hence, verifiable secret-sharing schemes can be avoided. Further, a (q, q) perfect secret-sharing scheme can be realized using simple bit-wise XOR operations. If each data bit is thought of as a separate secret, then each share is a single bit and the XOR of the q shares (or q bits) gives the encoded secret bit. In practice, XOR secret sharing can be implemented with word-wide operations for efficiency. Table 5 lists the computation times during secret sharing and secret recovery for a selection of (q, q) values for XOR secret sharing. Note that XOR secret sharing is also a perfect secret-sharing scheme. The only constraint compared to the general (k, q)-threshold scheme with $k < q$ is that all q shares must be recovered to reconstruct the secret. Compared with the computation times using Shamir's scheme (Table 2), the computation times using XOR secret sharing are much lower.

Mechanism 2: Use replication-and-voting to determine incorrect shares

Table 5: Computation times for XOR sharing (8 KB block)

(q, q)	Secret sharing	Secret recovery
$(5, 5)$	333 μs	35 μs
$(10, 10)$	732 μs	60 μs
$(20, 20)$	1.494 ms	140 μs

during reads: To detect incorrect shares that may be returned by malicious servers during reads, we propose that each share be replicated at enough servers such that if at least a threshold of servers returns the same share during a read, then that share is correct and can be used for the secret recovery computation. This is the traditional technique used for managing replicated data, which we apply for each share. If the number of malicious servers is denoted by b, then for each share at least $(2b + 1)$ responses must be received. The value returned by at least $(b + 1)$ servers is the correct value of the share being read.

Table 6: Computation times for voting out of 2b+1 responses to determine a share of size 8 KB. Measurements reflect the best case where there are no incorrect responses.

b	1	2	3	4	5
Computation Time (μs)	13.75	25	40	50	65

Table 6 gives the computation times for determining each share from $(2b + 1)$ responses, where b is the number of possibly malicious servers. Note that the numbers are given for each share. Hence, the computation time during secret recovery must now include the product of the time taken to determine each share from $(2b + 1)$ responses and the number of shares. The secret sharing computation time will remain unchanged as no additional shares are generated. The secret sharing and recovery computation times for XOR secret sharing along with voting for $b = 3$ are shown in

Table 7: Secret sharing and recovery computation times for XOR secret sharing with voting (8 KB block, b = 3).

(q, q)	Secret sharing	Secret recovery
$(5, 5)$	333 μs	235 μs
$(10, 10)$	732 μs	460 μs
$(20, 20)$	1.494 ms	940 μs

Table 7. Compared with the computation times of verifiable secret-sharing schemes (Table 3), the computation times of XOR secret sharing with voting are much lower and are in the same order of magnitude as those of the Rijndael encryption algorithm (Table 4).

Summarizing, perfect secret-sharing schemes can be used for fault-tolerant and secure distributed data storage by combining them with verifiable secret-sharing schemes. Using the computation latency of the Rijndael algorithm as the benchmark, we have shown that well-known verifiable secret-sharing techniques such as the combination of Feldman's scheme with Shamir's scheme are too slow to be used for large volumes of data. The computation overhead can be drastically reduced by using instead a (q, q) perfect secret-sharing scheme (namely, XOR secret sharing) along with replication-and-voting mechanisms. The computation times are comparable to those of the Rijndael algorithm. The next section describes how XOR secret sharing with replication-and-voting mechanisms can be combined.

4.5 Combining Secret Sharing and Replication

Our approach in this chapter for realizing a fault tolerant and secure data storage service is to use perfect threshold secret sharing for data confidentiality and to use replication-based mechanisms to manage each share for crash and Byzantine fault tolerance. We first describe a straightforward method of using this approach, called

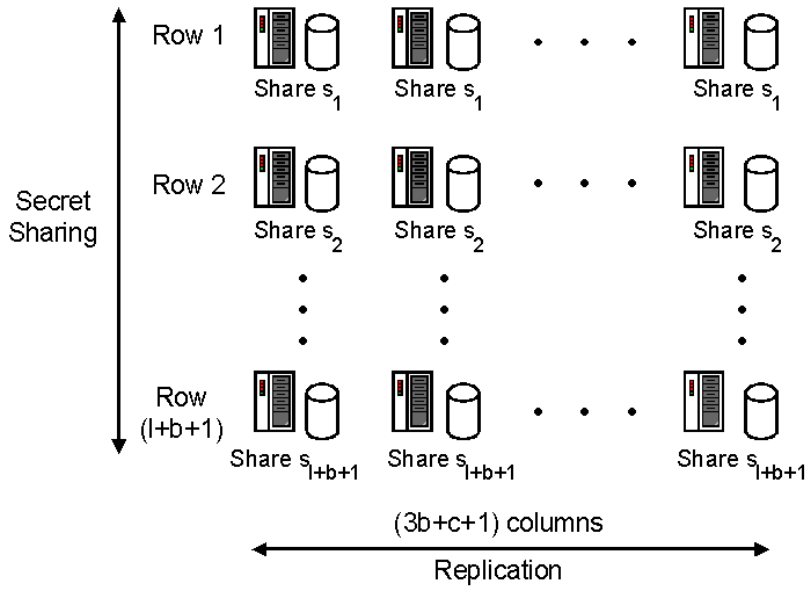

Figure 4: The *Direct Approach*: Servers are arranged in a logical grid having (l+b+1) rows, with at least (3b+c+1) servers in each row. Secret sharing is done across rows, with a distinct share assigned to each row. Shares are replicated along rows.

the *direct approach*, and show that it suffers from requiring a large number of storage servers. We then introduce the *GridSharing* framework, where a tradeoff between the number of servers required and the storage space needed at each server is achieved. This is a worthwhile tradeoff because storage space is cheap.

4.5.1 The Direct Approach

We consider a system consisting of N storage servers, where not more than l servers can be leakage-only faulty, not more than b servers can be Byzantine faulty, and not more than c servers can crash. The direct solution to this is to use a $(l+b+1, l+b+1)$-threshold perfect secret-sharing scheme. Each share is given to a distinct set of x servers. The setup can be envisioned as the N servers arranged in the form of a logical grid with $(l + b + 1)$ rows and x columns, as shown in Figure 4.

Servers in the same row store the same shares. The replication of shares is used to achieve crash and Byzantine fault tolerance. Data confidentiality is achieved using

secret sharing. The secret sharing is done across rows. Thus, $(l + b + 1)$ rows are required, with each share assigned to a distinct row. The compromise of any $(l + b)$ servers will give only up to $(l + b)$ shares to an adversary, but all $(l + b + 1)$ shares are needed to recover the secret.

When secrets are read and written, the shares are read and written using replication-based protocols. For the purposes of this and subsequent analyses in this chapter, we assume the following simple replication protocol. To write a secret S, the user generates $(l + b + 1)$ shares such that their bitwise-XOR gives the secret S. The user writes to each server its assigned share. Thus, in the example depicted in Figure 4, the user will write to each server in row 1 the share s_1, to each server in row 2 the share s_2, and so on.

When the secret S is to be read at a later time, the same user or a different user will need to only contact some set of servers to read all the shares. Consider how share s_1 is read in our example. The share s_1 is stored in row 1, which consists of x servers. The user needs to contact only $(2b + 1)$ of these servers to determine s_1, since only a maximum of b servers can be Byzantine faulty. The share s_1 returned by at least $(b + 1)$ servers must have been returned by at least one server that is not Byzantine-faulty, and therefore should be correct. The user must obtain at least $(2b + 1)$ responses to determine s_1, but up to $(c + b)$ servers can fail to return any response. Assuming clients connect to the servers over an asynchronous network so that they are unable to detect server failures, each share must be written to at least $((2b + 1) + (c + b)) = (3b + c + 1)$ servers for reads to be successful in the presence of b Byzantine failures and c crash failures in the system.

Thus, each share must be stored on at least $(3b + c + 1)$ servers. Thus, $x \geq (3b + c + 1)$, which gives $N \geq (l + b + 1)(3b + c + 1)$. Note that the given description for writes and reads is only an approach for a possible replication-based protocol to

manage the shares. We have overlooked the need for the use of timestamps, which are common to all the shares. All the shares must be written as part of a single write operation. The approach described is just sufficient to derive a lower bound on the number of servers required to store each share. This lower bound will change based on the assumptions on the system model and the kind of read-write semantics to be realized. The minimum number of servers needed to maintain each share is the only point in the design of the framework that is dependent on the choice of the replication protocol and its underlying assumptions.

Thus, to tolerate l leakage-only faults, b Byzantine faults, and c crash faults, at least $(l+b+1)(3b+c+1)$ servers are required for this approach. For $l = b = c = 2$, at least 45 servers are required. That is, only up to $6/45 = 13.3\%$ servers can be faulty. This is inefficient in terms of the number of storage servers required. However, the storage blowup at each server is one, as the size of each share is the same as the size of the encoded secret. Also, the bare minimum number of shares is generated, which is $(l + b + 1)$. Thus, the computation times during secret sharing (writes) and secret recovery (reads) at the clients are kept as low as possible.

In the next section, we describe the *GridSharing* framework, where we balance the strengths and the weakness of the *direct approach*. We trade off the number of storage servers required with the storage blowup at each server and the total number of shares generated for each secret.

4.5.2 The GridSharing Framework

Similar to the *direct approach*, the *GridSharing* framework consists of N servers, where not more than c servers can crash, not more than b servers can be Byzantine faulty, and not more than l servers can exhibit leakage-only faults. The N servers are arranged in the form of a logical rectangular grid with r rows and $\frac{N}{r}$ columns, where

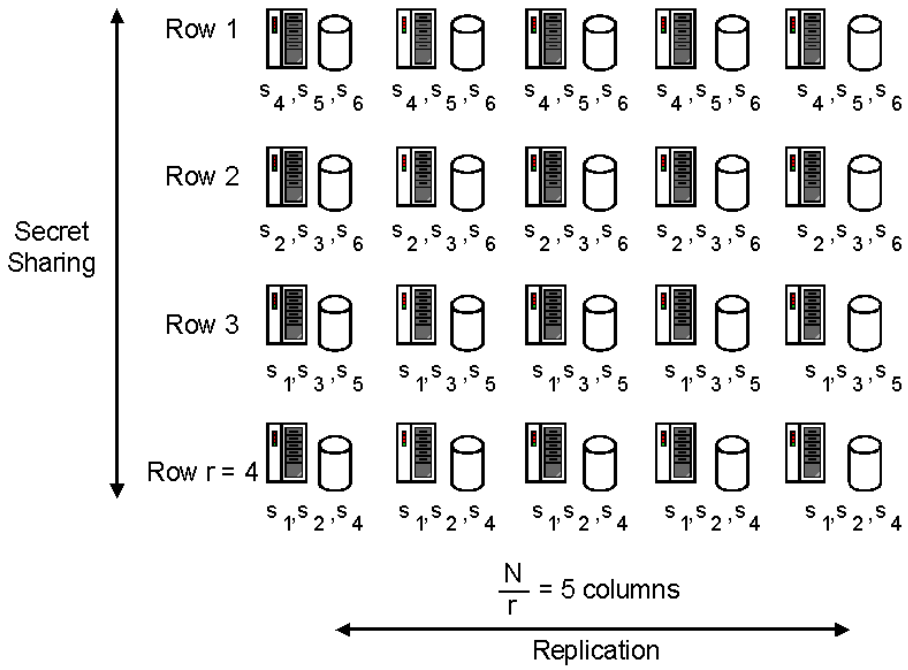

Figure 5: The *GridSharing* framework: N servers are arranged in a logical grid having r rows. Secret sharing is done across rows, and shares are replicated along rows. Setup shown for N = 20, l = 1, b = 1, and c = 6. Note that each server stores three shares.

for simplicity we assume that N is a multiple of r. The arrangement is depicted in Figure 5.

Servers in the same row store the same shares. Thus, tolerance to crash and Byzantine failures is achieved. Data confidentiality is achieved using secret sharing. The secret sharing is done across rows. Ito et al.'s [36] share assignment scheme is used to assign shares to the rows. Thus, as per the terminology used in Section 4.3.2, the r rows are the r participants among which shares are distributed. Since up to l servers can be leakage-only faulty (reveal their shares to an adversary) and up to b Byzantine-faulty servers can also do the same, shares from up to $(l + b)$ rows can be disclosed to an adversary. From Section 4.3.2, an $\left(\binom{r}{l+b}, \binom{r}{l+b}\right)$-threshold perfect secret sharing scheme can be used to tolerate $(l + b)$ faulty servers in r rows.

Figure 5 gives an example where $N = 20$ servers are arranged in a rectangular grid with $r = 4$ rows. If it is necessary to tolerate $b = 1$ Byzantine fault and $l = 1$ leakage-only fault, then a $\left(\binom{4}{2}, \binom{4}{2}\right) = (6, 6)$ XOR secret sharing scheme will have to be used. Assume a secret S is encoded into six shares $(s_1, s_2, s_3, s_4, s_5, s_6)$ such that $S = s_1 \oplus s_2 \oplus s_3 \oplus s_4 \oplus s_5 \oplus s_6$. That is, each bit in the secret S is the XOR of the corresponding bits in the shares $s_1, s_2, s_3, s_4, s_5, s_6$. Then, according to the share assignment function g given in Section 4.3.2,

Servers in row 1 get shares (s_4, s_5, s_6),

Servers in row 2 get shares (s_2, s_3, s_6),

Servers in row 3 get shares (s_1, s_3, s_5),

Servers in row 4 get shares (s_1, s_2, s_4).

The choice of Ito et al.'s share assignment scheme is motivated by the fact that each share is assigned to multiple rows. This is in line with our principle of using the replication of shares to achieve Byzantine and crash fault tolerance. Note also that, as in the *direct approach*, shares are replicated along rows. As argued in Section 4.5.1, each share must be stored on at least $(3b+c+1)$ servers. In the proposed framework, each share is assigned to $(r - (l + b))$ rows, and each row has $\frac{N}{r}$ servers. Thus, each share is stored at $(r - (l + b))\frac{N}{r}$ servers, and this must be at least $(3b + c + 1)$. Thus,

$$(r - (l + b))\frac{N}{r} \geq 3b + c + 1 \tag{1}$$

which gives

$$r \geq \frac{N(l + b)}{N - (3b + c + 1)} \tag{2}$$

Inequality 2 gives the smallest number of rows possible for the framework. Thus, r can vary in the range $\left[\frac{N(l+b)}{N-(3b+t+1)}, N\right]$. Also, r must be greater than $(l+b)$; otherwise, a Byzantine fault or a leakage-only fault in each row will give the adversary all the

shares to recover the encoded data. From Inequality 2, it is obvious that the lower bound on r is greater than $(l + b)$.

For a given l, b, c, and r, Inequality 1 can be rewritten as

$$N \geq \frac{3b + c + 1}{1 - \frac{l+b}{r}} \tag{3}$$

to give a lower bound on the number of servers N required. The lower bound is minimized for a given l, b, and c when r is at its maximum value, which is N. Substituting $r = N$ in Inequality 3 gives the following requirement for N for tolerating l leakage-only faults, b Byzantine faults, and c crash faults:

$$N \geq 4b + l + c + 1 \tag{4}$$

Thus, as the number of rows r is increased from $(l + b + 1)$ to $(4b + l + c + 1)$, the minimum number of servers required will decrease. When $r = (4b + l + c + 1)$, the smallest number of servers needed to tolerate b Byzantine, c crash, and l leakage-only faults will be reached. For $r > (4b + l + c + 1)$, there will be only one column, the number of servers N will be the same as the number of rows r, and N will increase with r.

4.6 Performance Analysis of GridSharing

4.6.1 Performance Metrics

This section defines some performance metrics, whose relation with the fault tolerance and security properties l, b, and c, and the number of rows r, are described in the remainder of this section.

- **min(N)**: The minimum number of servers required for a given l, b, c, and r. This is given by the smallest N satisfying Inequality 3, with N being a multiple of r.

- **#Shares**: The total number of shares generated per secret. For the proposed framework, #Shares = $\binom{r}{l+b}$.

- **Storage Blowup Per Server**: The ratio of the storage space taken at each server to the size of the data encoded. For the proposed framework, the storage blowup factor is $\binom{r-1}{l+b}$. Since we use the XOR secret sharing scheme, the size of a share is the same as the size of the secret.

- **Secret Sharing and Secret Recovery Computation Times**: The time taken to encode and decode shares of an 8 KB block of data. The secret recovery computation time is the sum of two components. The first component is the time taken to determine the correct (#Shares) shares from $(2b+1)$ responses for each share, where b is the Byzantine fault tolerance threshold. We assume the best case where there are no incorrect servers when evaluating this component. The second component is the time taken to compute the data block once the correct (#Shares) shares have been determined. The size of the data block and each share are 8 KB. The measurements were taken on a Pentium4 3GHz computer with 256 MB RAM running Linux 2.6.9. All measurements were performed in memory and involved no disk and network I/O.

4.6.2 Effect of Grid Dimension

For given security and fault tolerance thresholds l, b, and c, the performance metrics can be traded off against each other by varying the number of rows r in the framework. The secret sharing and recovery computation times are dependent on #Shares, which is dependent on r and $(l+b)$. The smaller the number of rows r, the fewer the number of shares (#Shares) and the lower the computation times during secret sharing and secret recovery. But if r is increased from $(l+b+1)$ to $(4b+l+c+1)$, from Inequality 3,

Table 8: Effect of increasing number of rows r on performance metrics when l = 2, b = 2, and c = 2

r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
				Secret Sharing	Secret Recovery
5	45	5	1	333 μs	160 μs
6	30	15	5	1.103 ms	490 μs
7	21	35	15	2.668 ms	1.150 ms
8	24	70	35	5.480 ms	3.020 ms
9	18	126	70	10.31 ms	6.276 ms

the minimum number of servers required will decrease. Thus, the number of rows affects min(N) and the secret sharing and recovery computation times in opposing ways. For $l = 2$, $b = 2$, and $c = 2$, the tradeoff space is given in Table 8.

Table 8 shows that increasing the number of rows from $(l + b + 1)$ reduces the minimum number of servers required for that configuration while increasing the number of shares, #Shares, needed to store each secret. The storage capacity required at each server thus increases with r. Increasing #Shares will also increase the computation overhead at the users during the secret sharing and secret recovery processes. The practical range of r is thus limited by the storage blowup and the computation overhead.

When there are five rows in the framework, each row gets a distinct share (which is, the *direct approach*). The number of shares (#Shares) generated is minimum, and the computation times are small. But 45 servers are required for this configuration. By having seven rows in the framework, the minimum number of servers required is lowered by more than half to 21 servers. For given fault tolerance and security thresholds, having fewer servers implies that a higher percentage of faulty servers is tolerated. Having fewer servers will also increase the manageability of the system. On the other hand, the storage blowup at each server increases by a factor of 15. Since storage cost is cheap, this is a worthwhile tradeoff. The computation times are

Table 9: Effect of increasing l on performance when b = 2, c = 2, and min(N) ≤ 35 servers

l	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	5	25	10	4	732 μs	310 μs
2	6	30	15	5	1.103 ms	490 μs
3	7	35	21	6	1.568 ms	706 μs
4	9	27	84	28	6.750 ms	4.084 ms
5	10	30	120	36	9.675 ms	6.120 ms

Table 10: Effect of increasing b on performance when l = 2, c = 2, and min(N) ≤ 35 servers

b	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	4	24	4	1	267 μs	80 μs
2	6	30	15	5	1.103 ms	490 μs
3	8	32	56	21	4.315 ms	2.740 ms
4	11	33	462	210	38.88 ms	37.41 ms
5	16	32	11440	6435	3.104 sec	2.319 sec

also at acceptable values when $r = 7$. Thus, the choice of the number of rows in the framework can be used to arrive at a suitable tradeoff point between the number of servers required, the storage blowup, and the secret sharing and recovery computation overheads.

Table 11: Effect of increasing c on performance when l = 2, b = 2, and min(N) ≤ 35 servers

c	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	6	24	15	5	1.103 ms	490 μs
2	6	30	15	5	1.103 ms	490 μs
3	6	30	15	5	1.103 ms	490 μs
4	7	28	35	15	2.668 ms	1.150 ms
5	7	28	35	15	2.668 ms	1.150 ms

4.6.3 Effect of Fault Thresholds Given N Servers

In this section, we assume that 35 data storage servers are available and investigate the relation between the fault tolerance and security thresholds l, b, and c and the performance metrics. We consider three cases. In each case, we fix two of the thresholds at two servers and increase the other threshold from one to five servers. Tables 9, 10, and 11 show the three different cases. For each combination of (l, b, c), we fix the number of rows such that the secret recovery computation time is the smallest possible for the given configuration. Since the secret recovery computation time decreases with increasing r, for the given (l, b, c), r is set to the smallest value ($r \geq \frac{N(l+b)}{N-(3b+c+1)}$) such that $\min(N)$ is not more than 35 servers.

From Table 9, increasing the leakage-only fault threshold l leads to a tolerable increase in the storage blowup per server, while the secret sharing and recovery computation times become high for $l \geq 4$ servers. The effect of increasing the Byzantine fault threshold b, as shown in Table 10, has a more adverse effect on performance. The storage blowup per server and the secret sharing and recovery computation times increase rapidly with increasing b. Thus, to achieve a very high performance with 35 servers, only a relatively small number of Byzantine failures can be tolerated.

On the other hand, the framework can accomodate more crash failures without any substantial performance impact, as shown in Table 11. Increasing the crash fault threshold from one to five servers leaves the performance metrics mostly unchanged. The storage blowup at each server is tolerable and the computation throughputs are maintained at acceptable levels.

The examples considered above demonstrate that the framework can tolerate crash failures with little performance impact, leakage-only faults with medium peformance impact, and a limited number of Byzantine faults. The maximum number of faults

Table 12: Effect of increasing l on performance when b = 2, c = 2, and secret recovery computation time $\leq$ 1.6 ms

l	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	6	18	20	10	1.494 ms	640 μs
2	7	21	35	15	2.668 ms	1.150 ms
3	7	35	21	6	1.568 ms	706 μs
4	8	40	28	7	2.109 ms	928 μs
5	9	45	36	8	2.742 ms	1.196 ms

Table 13: Effect of increasing b on performance when l = 2, c = 2, and secret recovery computation time $\leq$ 1.6 ms

b	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	6	12	20	10	1.494 ms	415 μs
2	7	21	35	15	2.668 ms	1.150 ms
3	7	42	21	6	1.568 ms	1.02 ms
4	8	64	28	7	2.109 ms	1.60 ms
5	8	144	8	1	592 μs	576 μs

that can be tolerated is given by Equation 4. Thus, given 35 servers, when $b = 2$ and $c = 2$, up to 24 leakage-only faults can be tolerated; when $l = 2$ and $c = 2$, up to 7 Byzantine faults can be tolerated; and when $l = 2$ and $b = 2$, up to 24 crash faults can be tolerated. However, practical limits on the secret sharing and recovery computation times and the storage blowup at each server are a more severe restriction on the actual range of faults that can be tolerated. Notice that, except for high values for the Byzantine fault threshold b, the secret sharing and recovery computation times are much smaller than the figures given for verifiable secret sharing in Table 3.

4.6.4 Effect of Fault Thresholds Given Restriction on Secret Recovery Computation Time

Since increasing l, and b in particular, can lead to a substantial increase in secret sharing and secret recovery computation times, as observed in Table 9 and Table 10,

Table 14: Effect of increasing c on performance when l = 2, b = 2, and secret recovery computation time $\leq$ 1.6 ms

c	r	min(N)	# Shares	Storage Blowup Per Server	Computation Time	
					Secret Sharing	Secret Recovery
1	7	21	35	15	2.668 ms	1.150 ms
2	7	21	35	15	2.668 ms	1.150 ms
3	7	28	35	15	2.668 ms	1.150 ms
4	7	28	35	15	2.668 ms	1.150 ms
5	7	28	35	15	2.668 ms	1.150 ms

we remove the requirement of having only 35 storage servers available, and instead impose the requirement that the secret recovery computation time for 8 KB of data must be less than 1.6 ms. The secret recovery computation time is important when reads are more frequent than writes, which is often the case. A secret recovery computation time of 1.6 ms for 8 KB of data is approximately six and eight times slower than the decryption time using the Rijndael encryption algorithm for key sizes of 32 bytes and 16 bytes respectively, as was shown in Table 4.

Similar to Section 4.6.3, we consider three cases. In each case, we fix two of the fault thresholds at two servers, and increase the other fault threshold from one to five servers. Tables 12, 13, and 14 show the three different cases. For each combination of (l, b, c), we fix the number of rows r that gives the smallest min(N) while maintaining the secret recovery computation time to be less than 1.6 ms. Restricting the secret recovery computation time limits the number of shares (#Shares) generated, which in turn keeps the storage blowup at each server reasonable. In Table 12, the minimum number of servers required (min(N)) shows a moderate increase with increasing l. When $l = 5$ servers, a total of 9 $(l + b + c)$ servers out of 45 servers are faulty. That is, up to 20% of the servers can be faulty (leakage-only, Byzantine, or crash), which should be acceptable. In Table 13, the minimum number of servers required (min(N)) increases rapidly with the Byzantine fault threshold b. Thus, the proposed framework

is suitable for tolerating a small number of Byzantine faults.

In Table 14, the computation throughputs and the storage blowup remain the same with increasing crash fault threshold c for the example considered. With 21 servers, up to two crash faults are tolerated, and with 28 servers, up to 5 crash faults can be tolerated. Note that with 5 crash faults, a total of 9 servers out of 28 servers can be faulty. That is, up to 32% of the servers can be faulty, which is a standard property of replica management protocols that tolerate only Byzantine faults. While in this example most of the faults are crash faults, the number of servers required is reasonable.

Thus, from Tables 12, 13, and 14, low secret recovery computation times can be achieved with acceptable requirements on the number of servers and the storage blowup at each server. As observed in Section 4.6.3, the number of servers required for tolerating crash and leakage-only faults is acceptable, while practical considerations will restrict the number of Byzantine faults that can be tolerated. Note that, in all the analyses, the number of rows was manipulated to arrive at the optimum configuration.

4.7 Discussion

While the *GridSharing* framework aims to decrease the computation overheads incurred during secret sharing and recovery, the storage blowup at each server is increased, which increases the communication overhead during reads and writes. Tables 15 and 16 show the computation and communication overheads during the secret sharing (writes) and secret recovery (reads) processes for encryption, verifiable secret sharing (VSS), and the *GridSharing* framework when the fault thresholds l, b, and c are all equal to one.

The total time taken during a write operation is composed of three parts - the

Table 15: Comparison between encryption, verifiable secret sharing, and GridSharing during writes for 8 KB of data. l = 1, b = 1, and c = 1.

Coding Scheme	**min(N)**	**# Shares**	**Storage Blowup Per Server**	**Overhead during Writes**			
				Secret Sharing	**Encryption (Secure Channels)**	**Comm. Time**	**Total**
Encryption	5	1	1	205 μs		3.277 ms	3.482 ms
VSS	5	5	4	728ms	1.23 ms	13.11 ms	742.34 ms
GridSharing #rows = 3	15	3	1	180 μs	3.28 ms	9.83 ms	13.29 ms
GridSharing #rows = 4	12	6	3	430 μs	7.995 ms	23.59 ms	32.02 ms
GridSharing #rows = 5	10	10	6	732 μs	13.53 ms	39.32 ms	53.58 ms
GridSharing #rows = 6	12	15	10	1.103 ms	26.65 ms	78.64 ms	106.39 ms
GridSharing #rows = 7	7	21	15	1.568 ms	24.6 ms	68.81 ms	94.98 ms

Table 16: Comparison between encryption, verifiable secret sharing, and GridSharing during reads for 8 KB of data. l = 1, b = 1, and c = 1.

Coding Scheme	**min(N)**	**# Shares**	**Storage Blowup Per Server**	**Overhead during Reads**			
				Secret Sharing	**Encryption (Secure Channels)**	**Comm. Time**	**Total**
Encryption	5	1	1	218.75 μs		1.966 ms	2.185 ms
VSS	5	5	4	747.5 ms	820 μs	7.864 ms	756.18 ms
GridSharing #rows = 3	15	3	1	61.25 μs	2.05 ms	5.898 ms	8.01 ms
GridSharing #rows = 4	12	6	3	122.5 μs	4.1 ms	11.796 ms	16.02 ms
GridSharing #rows = 5	10	10	6	207.5 μs	6.765 ms	19.66 ms	26.63 ms
GridSharing #rows = 6	12	15	10	321.25 μs	10.045 ms	29.49 ms	39.86 ms
GridSharing #rows = 7	7	21	15	469.75 μs	14.76 ms	41.288 ms	56.25 ms

secret sharing operation, encryption of the shares to establish secure channels between the client and the servers, and the communication time. Similarly, the total time taken during a read operation consists of the communication time in getting the required number of shares, decrypting the shares from the secure channels, and then recovering the secret. For the communication time, it is assumed that the network bandwidth between the client and the servers is 100 Mbps. Note that our read / write protocol is a very simple one where a client writes to all servers and reads from the required number of servers. Timestamps and the use of Message Authentication Codes for providing message integrity during communication have been overlooked. It is also assumed that the client reliably gives each server its share(s) during writes, thus eliminating the need to implement a reliable broadcast protocol. The figures given serve only to compare between GridSharing, verifiable secret sharing schemes (namely, the combination of Shamir's and Feldman's scheme), and encryption.

The figures given for encryption do not take into account the overheads due to the storage and retrieval of cryptographic keys. The encrypted data need not be re-encrypted to achieve secure channels. The read and write latencies are thus very small. Only the minimum number of servers ($= (3b + c + 1)$) are required, and the storage blowup at each server is one. Data storage using the encrypt-and-replicate storage model is hence an attractive option when performance is critical. However, the security of the model relies on the secure maintenance of the cryptographic keys.

For VSS, the number of servers required is only 5 ($= (3b + c + 1)$). A $(3, 5)$-threshold Shamir's scheme is used, because up to two servers ($= (l + b)$) can leak shares to an adversary. The write and read latencies of VSS are over 213x and 346x slower than those of encryption. The secret sharing and recovery computation overheads account for over 98% of the total write and read latencies. In GridSharing, the secret sharing and recovery computation overheads are decreased substantially,

while the communication overheads are increased. However, the overall write and read latencies for GridSharing are still much less than that of VSS. When the number of rows r is set to 7 in GridSharing, the write and read operations are over 7x and 13x faster than that of VSS, respectively. The number of servers required is only two more than that of VSS, but the storage blowup at each server is 15. Decreasing r in *GridSharing* decreases the read and write latencies and the storage blowup at the expense of requiring more storage servers. When $r = 3$, the write and read latencies are comparable to those of encryption, but three times more storage servers are required.

The increased storage blowup in GridSharing should not be a limitation, as storage space is cheap. The fact that large amounts of inexpensive, surplus storage are available has been exploited in other applications, such as in [59], where the surplus storage space is used to store different versions of objects for subsequent intrusion diagnosis and recovery.

Finally, we would like to note that the communication overheads when using replication-based protocols can be reduced using other techniques. In [38], the use of cryptographic hashes when reading replicated data has been shown to significantly reduce the read latency. In [63], the tradeoff between computation and communication overheads for several lossless compression algorithms is investigated. Cryptographic hashes and compression algorithms reduce communication overheads while increasing the computation overheads, which reinforces the need for reducing the computation overheads during the secret sharing and recovery processes.

4.8 Conclusions

This chapter presents a novel approach for realizing a secure and fault tolerant data storage service using the *secret sharing* data storage model. Key highlights of this

work are:

- Verifiable secret sharing schemes are typically used with perfect secret sharing schemes to achieve Byzantine fault tolerance. We show that verifiable secret sharing schemes incur substantial computation overheads, and are much slower than the Rijndael encryption algorithm.

- We use an (n, n)-threshold perfect secret sharing scheme, namely the XOR secret sharing scheme, for confidentiality, and manage each share using replication-based protocols for Byzantine and crash fault tolerance. The computation overheads are reduced drastically when compared to verifiable secret sharing schemes, but additional servers and storage capacities at each server are required. An example where the secret recovery computation time was only up to 6 to 8 times slower than the Rijndael decryption algorithm was given.

- We present an architectural framework, called *GridSharing*, whose dimension can be varied to tradeoff between the number of servers required, and the storage blowup and secret sharing and recovery computation times. This property was shown to be valuable in arriving at optimum configurations for different fault thresholds.

- For secret recovery computation times that are 6 to 8 times slower than Rijndael decryption, we show that our proposed framework provides good fault tolerance to leakage-only and crash faults with acceptable overheads. However, in practice, resource limitations place a restriction on the number of Byzantine server failures that can be tolerated.

- A rough comparison of the overheads, including read and write latencies, between encryption-with-replication, verifiable secret sharing (VSS), and *GridSharing* was given. Since *GridSharing* incurs a higher storage blowup at each server, the read and write communication overheads are higher than with VSS. Despite this, GridSharing has lower overall write and read latencies than VSS. Write and read latencies comparable to storing data using private-key encryption schemes can be achieved at the expense of requiring a greater number of storage servers.

CHAPTER V

PERIODIC SHARE RENEWAL FOR THE GRIDSHARING FRAMEWORK

5.1 Problem Statement

In the mobile adversary model, there can be up to b Byzantine-faulty servers and up to l leakage-only faulty servers in every epoch. Byzantine and leakage-only faulty servers can reveal their stored shares to the mobile adversary. Thus, after sufficiently many epochs elapse, the mobile adversary may be able to obtain all the necessary shares to decode the stored data, thus violating the confidentiality property of the data storage service.

To maintain the confidentiality of the encoded data under the mobile adversary model, the shares of all data objects must be "renewed" periodically. The renewal process produces a new encoding of the shares. The attacker will thus not be able to obtain sufficient shares of the same encoding, thereby preserving the confidentiality of the encoded data.

For example, suppose Bit D is stored as the three share bits d_1, d_2, and d_3 such that $D = d_1 \oplus d_2 \oplus d_3$. If $l = b = 1$, then the adversary can learn two of these shares, say d_1 and d_2. In the next epoch, the encoding is changed to d'_1, d'_2, and d'_3 such that $D = d'_1 \oplus d'_2 \oplus d'_3$. The attacker may now learn d'_2 and d'_3. The attacker cannot decode

Bit D, as he needs to also know either d_3 or d'_1. The confidentiality of Bit D is thus maintained in the mobile adversary model by changing its encoding (or renewing its shares) in every epoch.

The share renewal process must be scalable with the large amounts of stored data. The GridSharing framework already helps in this regard via the use of XOR operations for secret sharing.

In the following sections, we describe the algorithm and protocol followed by experimental results.

5.2 The Share Renewal Algorithm

The share renewal algorithm is developed in the context of the GridSharing framework, described in Chapter 4. In the GridSharing framework, the servers are arranged in the form of a grid which has r rows. The secret sharing and share assignment is done across the r rows. XOR secret sharing is used, in which each data bit is encoded into several random *share bits* such the XOR of all these share bits gives the encoded data bit.

Since up to l servers can be leakage-only faulty and up to b servers can be Byzantine faulty in every epoch, the shares from up to $(l+b)$ rows can be revealed to the mobile adversary in an epoch. The share renewal algorithm to renew the share bits is run every epoch. Since the shares of large amounts of data need to be renewed, the share renewal process may run for a significant duration of an epoch. While the share renewal process is in progress in a new epoch, the attacker may compromise servers in a different set of $(l + b)$ rows and learn the share bits of the encoding used in the previous epoch. The GridSharing framework must therefore be designed to tolerate $2l$ leakage-only faults and $2b$ Byzantine faults.

The GridSharing framework uses an (n, n) XOR secret sharing scheme, where n is

given by $\binom{r}{2l+2b}$. All n shares are needed to decode the data, but the knowledge of any $(n-1)$ shares gives no information on the encoded data. Each data object is encoded into n shares and stored at the servers. Denote by S_1, S_2, ..., S_{n} the n shares of data object A. That is, S_i contains the i^{th} share of the encoding of each bit of data object A. To generate a new encoding of A, n random shares R_1, R_2, ..., R_n must be generated so that their bitwise-XOR gives object 0 in which each bit is 0. Thus, when a user reads A after the share renewal, he will read $(S_1 \oplus R_1) \oplus (S_2 \oplus R_2) \oplus ... \oplus (S_n \oplus R_n) = (S_1 \oplus S_2 \oplus ... \oplus S_n) \oplus (R_1 \oplus R_2 \oplus ... \oplus R_n) = A \oplus 0 = A$.

To generate a large amount of a random share, the RC4 stream cipher is used with a random string as the encryption key. The encryption key to generate the i^{th} share is given by SHA1(RStrng_i||Object Name), where RStrng_i is a random string and || denotes concatenation. To ensure that the random shares are the shares of 0, $(n-1)$ such random shares are generated using $(n-1)$ random strings to seed the RC4 cipher, and the n^{th} share is given by the bitwise-XOR of these $(n-1)$ shares. This procedure is shown in Figure 6.

The servers that store the n^{th} share must thus be made aware of the $(n-1)$ random strings used to generate the $(n-1)$ random shares. If one of these servers is leakage-only faulty or Byzantine faulty, then the adversary can learn the random sharings of bit 0 used for the share renewal, and the confidentiality of the stored data will be broken. To overcome this possibility, the share renewal algorithm is run n times. The j^{th} run involves computing the j^{th} share of a random encoding of bit 0 by XORing the other $(n-1)$ shares, that is, $R_j = R_1 \oplus ... \oplus R_{j-1} \oplus R_{j+1} \oplus ... \oplus R_n$.

The adversary can compromise up to $(l+b)$ rows while the share renewal is in progress. If, in the GridSharing setup, Share i is assigned to one of these $(l+b)$ rows, then the adversary can learn the random sharings of bit 0 generated in run i of the share renewal algorithm because computing Share i will require knowledge of all the

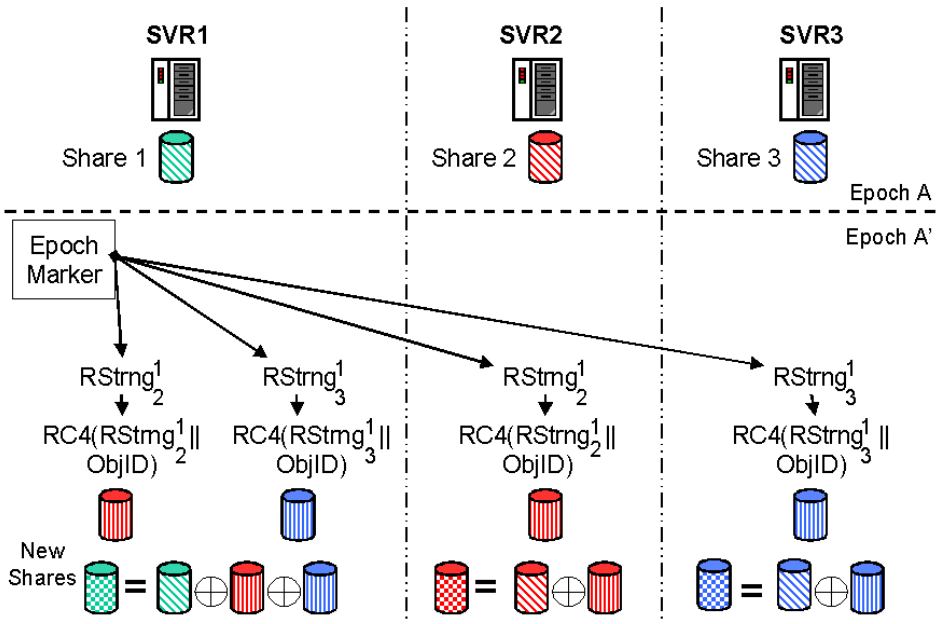

Figure 6: The Epoch Marker generates random strings which are used to generate the encryption keys for the RC4 stream cipher. The stream cipher is used to generate large blocks of random shares. One of the share holders (SVR1) must be made aware of all the random strings so that it can generate the appropriate share so that the XOR of all the random shares is zero. The share renewal algorithm is thus run n times, where n is the number of shares.

other shares. However, not all n shares are assigned to a group of $(l + b)$ rows. If Share i' is not assigned to these $(l+b)$ rows, then the adversary will be unable to learn all the random sharings of bit 0 generated in run i' of the share renewal algorithm.

The confidentiality of the encoded data is thus maintained under the mobile adversary model. The next section describes the protocol used during share renewal.

5.3 Share Renewal Protocol

Our proposed data store architecture has an *Epoch Marker* that divides time into *epochs*, and notifies the servers the start of a new epoch. The Epoch Marker is assigned the responsibility of generating the random strings used to seed the RC4 stream ciphers during the share renewal process. The protocol flow at the start of a

new epoch is shown in Figure 7.

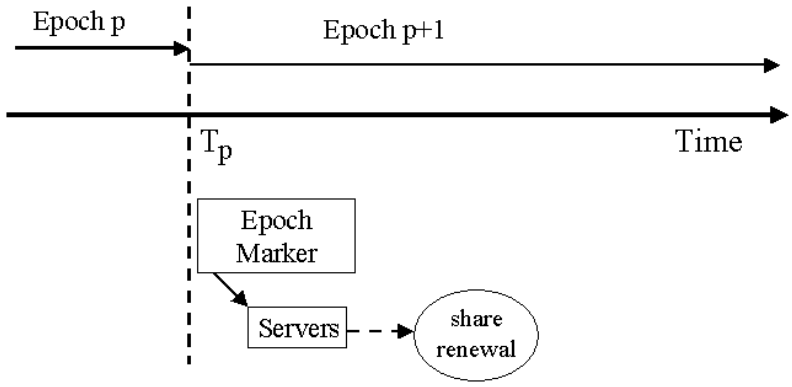

Figure 7: Servers start running the share renewal algorithm upon learning the start of a new epoch and the random strings for share renewal from the Epoch Marker.

Denote the set of random strings generated for the j^{th} run of the share renewal algorithm by $R^j = \{RStrng_1^j, ..., RStrng_{j-1}^j, RStrng_{j+1}^j, ..., RStrng_{\text{n}}^j, \}$. The Epoch Marker generates these random strings, and then notifies all the servers the start of a new epoch. The notify message for a server contains the random strings the server needs to perform the n share renewal runs. Thus, all servers that store Share i will receive from the Epoch Marker the random strings

$$RStrng_i^1, ..., RStrng_i^{i-1}, (R^i), RStrng_i^{i+1}, ..., RStrng_i^n.$$

We assume that the Epoch Marker is always fault-free.

The servers use the random strings to seed the RC4 stream ciphers and renew the shares they store for each data object. If the shares of a data object are not yet renewed and a read request from a user is received, then the shares are renewed and then sent to the user.

5.4 Experimental Analysis

This section presents an experimental analysis of the share renewal algorithm for the GridSharing framework. For experimentation purposes, we assume that the length of an epoch is one day. The shares of all the stored data objects must be renewed within this time. The various configurations of the GridSharing framework differ in the number of shares that are generated. The more the number of shares, the more time it takes to renew the shares and consequently less data can be stored, as the shares have to be renewed within the duration of an epoch.

We assume that up to one server can be leakage-only faulty and up to one server can be Byzantine faulty in an epoch, and up to one server can crash throughout the lifetime of the system. The GridSharing framework must thus be designed to tolerate 2 leakage-only faulty servers, 2 Byzantine faulty servers, and 1 crash fault.

The experimental analysis was performed in the context of a document storage system. The share renewal is done for preserving the confidentiality of the latest versions of all the stored documents. All of our experiments were run on Emulab [2]. The servers were Dell Poweredge 2850 machines consisting of 3.0 GHz 64-bit Xeon processors, 2 GB RAM, and 146GB 10000 RPM SCSI disks. The servers were not processing any reads and writes during these experiments.

Table 17: Effect of increasing number of rows r on the share renewal rate when up to one server can be leakage-only faulty and up to one server can be Byzantine faulty in an epoch, and up to one server can crash in the system lifetime.

r	**N**	**# Shares**	**Storage Blowup Per Server**	**Renewal Rate 1 MB Docs / s**	**# Docs Renewed In 1 Day**
5	40	5	1	8.92	770K
6	24	15	5	7.45×10^{-1}	64K
8	16	70	35	2.33×10^{-2}	2K
12	12	495	330	3.48×10^{-4}	30

Table 17 gives some possible configurations of the GridSharing framework, along

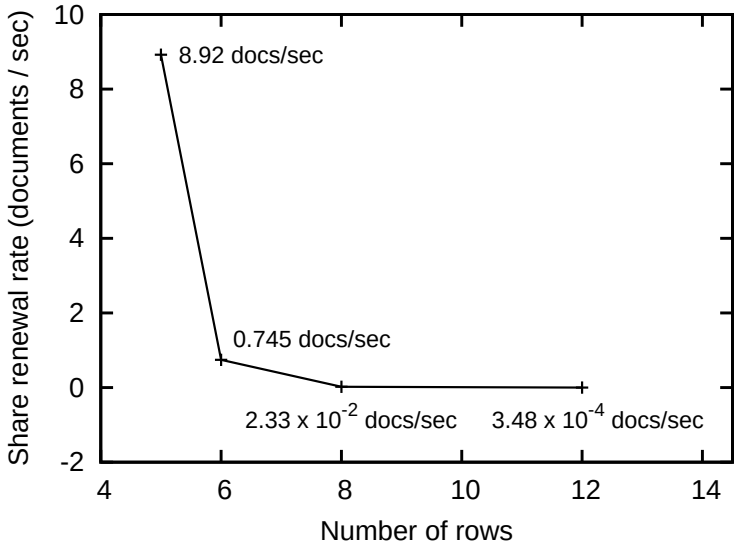

Figure 8: Share renewal rate vs. the number of rows in the GridSharing framework

with some performance metrics. The sizes of all the stored documents are set to 1 MB. As the number of rows is increased, fewer storage servers are required, but the number of shares generated and the storage blowup at each server increase. A server will take more time to renew its shares of a document, resulting in lower rates of share renewal. Figure 8 shows the share renewal rate vs. the number of rows r.

If the epoch length is assumed to be 1 day (86,400 seconds), then the number of documents whose shares can be renewed in a day is given in Table 17. Since the size of each document is set to 1 MB, for $r = 5$, over 770 GB of documents can be stored in a proactively-secure data store. When $r = 8$, only 2 GB of documents can be stored. The right choice of r is a tradeoff between the amount of data that can be proactively-secured and the number of servers required.

5.5 Conclusions

This chapter presents an algorithm for share renewal in the GridSharing framework. The share renewal process generates a new encoding for the stored data objects.

The share renewal process is run in every epoch. If up to l servers can be leakage-only faulty and up to b servers can be Byzantine faulty in every epoch, then the GridSharing framework must be designed to tolerate $2l$ leakage-only faults and $2b$ Byzantine faults. The servers use RC4 stream ciphers to generate random shares. The cryptographic keys used in the RC4 cipher are derived from random strings generated by the Epoch Marker.

The algorithm was experimentally evaluated on the Emulab cluster, in the context of a document storage service. If up to one server can be leakage-only faulty and up to one server can be Byzantine faulty in an epoch, and up to one server can crash, then the number of 1 MB-sized documents that can be proactively-secured varies from as high as 770K (770 GB) to as low as 30 (30 MB) documents, depending on how the GridSharing framework is configured. The independent parameter used to configure the GridSharing framework is the number of rows. The number of shares needed to store documents increases with more rows, thus reducing the number of documents whose shares can be renewed in an epoch. However, increasing the number of rows reduces the number of servers that are required. Thus, the right choice of the number of rows is a tradeoff between the amount of data that can be proactively-secured and the number of servers required.

CHAPTER VI

PERIODIC INTEGRITY VERIFICATION AND RESTORATION

6.1 Problem Statement

The data stored at a Byzantine-faulty server can be corrupted arbitrarily by an adversary. In the mobile adversary model, the Byzantine faults may move from one server to another every epoch. For a server that is not faulty in the current epoch but was Byzantine-faulty in an earlier epoch, its stored data will remain in the corrupted state. Thus, after sufficient epochs elapse, every server could be storing corrupted data.

To a client reading a stored data object, a fault-free server returning corrupted data may be indistinguishable from a Byzantine-faulty server behaving arbitrarily. However, the read-write protocols can tolerate only b Byzantine faults in N storage servers.

This problem is illustrated in Figure 9. Consider a system where the read-write protocols can tolerate only one Byzantine fault. In Epoch 0, all the servers are fault-free. In Epoch 1, Server 1 becomes Byzantine-faulty and its stored data is arbitrarily corrupted. Clients will still be able to read the correct data assuming the read-write protocols can tolerate a maximum of one Byzantine fault. In Epoch 2, the Byzantine

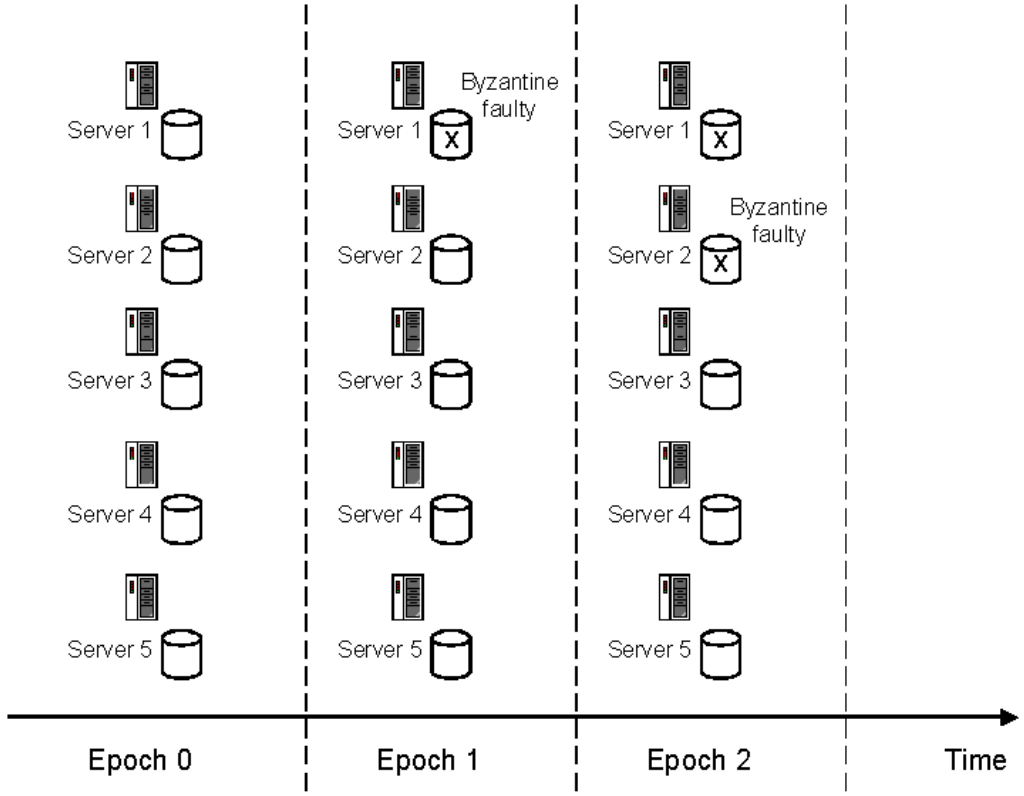

Figure 9: The mobile adversary in action: Eventually, the data stored at all the servers may be corrupted.

fault has moved from Server 1 to Server 2, and the data stored at Server 2 is corrupted. Two servers can now return corrupted data during reads, and the correctness property of the read-write protocols may not hold.

In the following sections, we describe some of the related works followed by the solution and experimental results.

6.2 Related Work

A well known filesystem integrity checker is Tripwire [4]. It is used to secure the integrity of the files stored in a computer. It is first used to create a database of filenames, their hashes and other attributes. The database, along with the tripwire

executable, is stored on a read-only medium so that an attacker compromising the computer cannot modify them. Tripwire can then later check if any files have been corrupted by comparing their attributes and hashes with the database. Tripwire is thus an intrusion detection tool that detects compromises after they have occurred. It does not include a mechanism to restore corrupted files from backups.

Our approach is similar to Tripwire, except that backups of the files and the database are available as replicated copies on other servers. Thus, we do not require a read-only medium, the files and the database can be modified by authorized users, and our approach is also able to restore corrupted files. Similar to Tripwire, our approach can also detect corrupted data and thus detect if a server was compromised in the last epoch.

Proactive recovery in a Byzantine fault-tolerant system was considered in [21]. The paper describes how replicas can periodically verify and recover their state. In [35], a technique to detect and repair corrupted shares for data stored at servers using secret sharing is given. Both [21] and [35] deal with only small amounts of data. We address the problem of scaling the integrity maintenance process to large amounts of data in a read-write storage system.

The lazy verification approach in PASIS [5] allows servers to detect and repair partial writes by faulty clients in a Byzantine-faults tolerant distributed data storage system. It is not clear if the lazy verification approach can be used to detect and repair corrupted *objects*, because an adversary can erase or create spurious objects at compromised servers. Also, because of their asynchronous system model, it is not clear how their approach can be proved to be secure in the mobile adversary model.

6.3 Solution Approach

The integrity of the stored data can be maintained in the mobile adversary model if the servers run some procedure periodically to detect and repair data corruptions.

There are two types of data stored at the servers: 1. *userdata*, and 2. *metadata*. The *userdata* is the actual data stored by the users of the storage service, while the *metadata* is information on the stored userdata. Types of metadata of particular interest for data integrity are *object* identifiers (such as filenames) and the checksum of the contents of the data objects.

The proposed solution consists of two steps.

Step 1: In every epoch, all the servers that are not Byzantine or crash faulty arrive at a consistent set of metadata for the userdata stored till the last epoch.

Step 2: Next, the non-faulty servers verify if their stored *userdata* match the metadata (determined in Step 1) and repair any corrupted *userdata* by reading it from other servers.

After the servers finish the above steps in the current epoch, the integrity of the data stored till the last epoch is verified and restored at all non-faulty servers. Servers that were Byzantine faulty in the previous epoch will be free of data corruptions; only servers that are Byzantine faulty in the current epoch may have their stored data corrupted. Steps 1 and 2 impose some requirements on the write protocol, as writes can straddle epoch boundaries. These requirements are given in Section 6.4.3.

Having the non-faulty servers arrive at consistent metadata ensures that stored objects are not deleted and spurious objects are not created. In addition, we require that a stored object's metadata contain the hash (checksum) of the contents of the object. This is required to detect arbitrary data corruptions at a server.

The proposed two-step solution approach is applicable in a variety of distributed data storage systems. The only requirement is that *all* the non-faulty servers are able to arrive at a set of consistent metadata. The *userdata* itself can be stored using secret sharing techniques, such as the GridSharing framework, or erasure codes. Further, the *userdata* is not required to be replicated at all the servers. For example, storage systems that use quorums for managing *userdata* can be adapted to have the metadata replicated at all the storage servers.

6.4 Protocol for Periodic Integrity Verification and Restoration

To counter the mobile adversary, we must have the servers run some distributed protocol periodically to repair data corruptions from earlier epochs. We call this protocol the *Periodic Integrity Verification and Restoration* (PIVR) protocol.

We develop this protocol in a system where the userdata and their metadata are replicated at all the storage servers. The storage servers store *data objects*. A write to a data object stores a *new version* of the data object. Versions of a data object are distinguished using *version numbers*. Versions numbers are considered a part of the metadata of the data object.

Our proposed data store architecture has an *Epoch Marker* that divides time into *epochs*, and notifies the servers the start of a new epoch. The exact flow of events at the start of a new epoch is shown in Figure 10.

A server, upon being notified the start of a new epoch by the Epoch Marker, updates its view of the epoch number and then starts running the PIVR protocol. Protocol PIVR consists of the following steps:

PIVR_Step1: Metadata check and repair: Servers execute this step to arrive at a consistent set of metadata for the userdata stored till the last epoch.

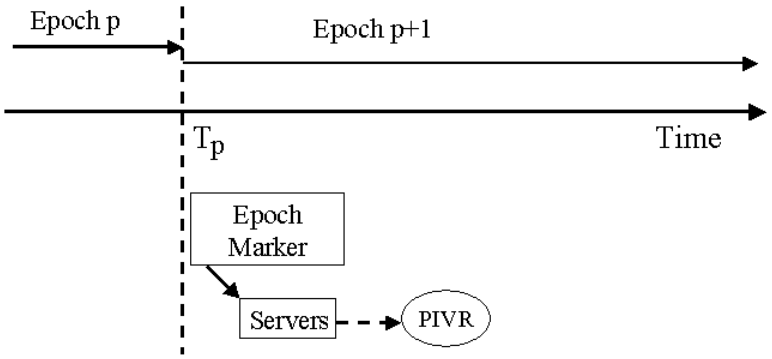

Figure 10: Servers start running the PIVR protocol upon learning the start of a new epoch from the Epoch Marker.

PIVR_Step2: Userdata check and repair: The stored userdata are checked against the metadata determined in PIVR_Step1 to detect corruptions. Corrupted userdata are repaired by reading the correct contents from other servers.

In a fully-replicated distributed data store, Protocol PIVR requires that the write protocols satisfy certain properties, given in Section 6.4.3. The synchronous system model is used for executing Protocol PIVR. This is required for Protocol PIVR to complete in an epoch. Thus, servers use the reliable synchronous network for sending and receiving protocol messages. The above two steps, PIVR_Step1 and PIVR_Step2, are explained below:

6.4.1 PIVR_Step1: Metadata Check and Repair

The pseudocode for the protocol for PIVR_Step1 is given in Figure 11. Each server first arrives at a list of locally-stored objects (userdata) and their metadata. The metadata includes the hash of the contents of the data objects. The servers then check against each other to determine the "correct" list. Since up to b servers can be Byzantine faulty in the previous epoch and b servers can be Byzantine faulty in the current epoch, the list reported by at least $(2b+1)$ servers is from at least one server

1. Arrive at a sorted list L_i of locally-stored data objects (userdata) and their metadata.
   Each entry in the list is of the form Object_ID||Metadata.
   Metadata includes the hash of the contents of the data object.
   The list is sorted on the object IDs.
2. Compute hash H_i of List L_i.
3. Request hash of lists from all the other servers.
4. Wait for response from all the other servers, or until timeout.
5. Determine hash H returned by at least $(2b + 1)$ servers, including itself.

6a. If $H_i = H$, List L_i is the correct list.

6b. If $H_i \neq H$, read the correct list from a server that returned H.

7. Delete from local storage all objects that are not present in the correct list.
8. End.

Figure 11: Pseudocode for PIVR_Step1 of the Periodic Integrity Checking and Verification Protocol.

that is not faulty in the current and previous epochs and is hence the correct list.

To make the reading and comparing of lists efficient (Step 3 in Figure 11), each server requests other servers for the *hash* of their lists instead of their entire lists.

6.4.2 PIVR_Step2: Document Integrity Checking and Repair

Having arrived at the correct list of object IDs and their hashes, a server proceeds to check if it has the data objects corresponding to each entry in the list. The integrity of the contents of an object are checked against the hash contained in the metadata for that object.

If the data object corresponding to an entry in the list does not exist, then the server reads the object off another server that reported the correct hash in PIVR_Step1. To safeguard against a malicious server that reported the correct list but may now return an incorrect data object, the server must compute the hash of the object read and check if it matches the hash contained in the metadata for that object. If the hashes don't match, the server reads the object off another server that reported the correct hash, until the hashes match.

6.4.3 Requirements on Writes

In a read-write fully-replicated distributed data store, Protocol PIVR requires that the write protocols satisfy certain properties, listed below:

PIVR_WProp1: Writes update the metadata at all the servers that are not Byzantine or crash faulty. The metadata that is updated includes the checksum of the new userdata included in the write request.

PIVR_WProp2: Writes store the contents of the new version at at least $(b + c - f_c + 1)$ servers that are not Byzantine or crash faulty, where f_c is the number of servers that have crashed.

PIVR_WProp3: Servers that are not Byzantine or crash faulty in the previous and current epochs can consistently identify the userdata stored till the last epoch.

PIVR_WProp4: Each server that is not Byzantine or crash faulty can safely conclude, at some point in the current epoch, that it has received all metadata updates due to writes from the previous epoch.

In a fully-replicated data store, the metadata and the userdata are stored at all the servers. PIVR_WProp1 is required, otherwise each server executing PIVR_Step1 will have to *merge* the metadata from all servers. A server must finish executing PIVR_Step1 as quickly as possible, so that the remainder of the epoch can be devoted to PIVR_Step2. A Byzantine-faulty server can report an impractical amount of metadata and cause resource exhaustion during this merge process.

PIVR_WProp2 requires writes to store the userdata at at least $(b + c - f_c + 1)$ servers. The remaining servers can update themselves with the corresponding userdata as part of PIVR_Step2 in the next epoch. A necessary requirement for this is that there is at least one fault-free server in the next epoch that has the correct

userdata. Among the servers that stored the userdata included in a write, b servers can become Byzantine faulty and c servers can crash in the next epoch.

PIVR_WProp3 is required because the PIVR protocol verifies and restores the integrity for the userdata stored till the last epoch. Satisfying this property is not trivial because the servers may not receive at the same time the notification of a new epoch from the Epoch Marker.

PIVR_WProp4 is required so that a server knows when it should start running the PIVR protocol in a new epoch.

In Chapter 8, a document repository realized using read-write protocols that satisfy the above properties is described.

6.4.4 Requirement on Minimum Number of Servers

Theorem 1 *In every epoch, the servers that are not Byzantine or crash faulty will, after running Protocol PIVR, store the correct metadata and the correct userdata contents for the userdata stored till the last epoch provided the number of servers is at least* $(4b + c + 1)$.

Proof: Consider the run of PIVR_Step1 in the second epoch of its lifetime. A server, as part of PIVR_Step1, will first read its local storage to arrive at the metadata for the userdata stored till the last epoch, which in this case is the first epoch. Because the writes are assumed to have properties PIVR_WProp1, PIVR_WProp3, and PIVR_WProp4, all servers that are not Byzantine or crash faulty in the first and second epochs will arrive at the same metadata for the userdata stored in the first epoch. We call this list the "correct" list. Servers that are Byzantine or crash faulty in either the first or the second epoch may not arrive at the correct list, as their local data storage may have been corrupted arbitrarily by the adversary. Thus, at least

$(N-2b-c)$ servers will have the correct list, as up to b servers can be Byzantine-faulty in every epoch and up to c servers can crash during the system lifetime.

A server determines if its list is correct by first reading the lists from all the servers, including itself. Since up to $2b$ servers may return a corrupted list, the list returned by at least $(2b+1)$ servers is guaranteed to be from at least one server that was not Byzantine faulty in Epochs 1 and 2. Thus, $(N-2b-c)$ must be at least $(2b+1)$, or N must be at least $(4b+c+1)$. The use of the synchronous system model ensures that a server will timeout waiting for responses from servers that do not respond (due to crash or Byzantine failures) and complete PIVR_Step1.

Once a server has determined the correct list, it starts executing PIVR_Step2. PIVR_Step2, together with Property PIVR_WProp2 of writes, ensures that all servers that are not Byzantine or crash faulty in the current epoch will be free of local data corruptions. Only servers that are Byzantine faulty in the current epoch may store corrupted data.

Thus, at least $(4b+c+1)$ servers are required in the system. To extend the analysis to subsequent epochs, we note that Protocol PIVR is executed by all servers that are not Byzantine or crash faulty in every epoch. Thus, when performing PIVR_Step1 at the start of the third epoch, all servers that were neither Byzantine nor crash faulty in Epochs 2 and 3 will arrive at the same list of objects and their metadata. The proof follows by induction. ∎

6.5 Experimental Analysis

This section presents an experimental analysis of Protocol PIVR to study its scalability with respect to the amount of stored data and the amount of corrupted data at compromised servers. For experimentation purposes, we assume that the length of an epoch is one day. Protocol PIVR must thus complete in a shorter time.

The experimental analysis was performed in the context of a document storage system. Users can upload and download documents. When a document is uploaded, a new *version* of the document is stored. Protocol PIVR is used to protect the integrity of the latest versions of all the stored documents.

All of our experiments were run on Emulab [2]. The servers were Dell Poweredge 2850 machines consisting of 3.0 GHz 64-bit Xeon processors, 2 GB RAM, and 146GB 10000 RPM SCSI disks. The machines were connected to each other over a 1 Gbps switched Ethernet LAN. Each document is 1 MB in size. The larger the document size, the longer it will take to complete PIVR_Step2, as this involves computing the hash over the document contents and reading it off another server if the contents are found to be corrupted. PIVR_Step1 will however be unaffected by the document size if the hash of the document is updated in the metadata list at the time of the write itself.

6.5.1 Scalability of PIVR_Step1

We studied the scalability of the protocol for PIVR_Step1 (given in Figure 11) w.r.t. the number of documents stored in the system. This step first requires servers to arrive at a sorted list of locally-stored document names and their highest versions. In our implementation, all versions of a document *docname* are stored in the directory `$DATADIR/docname`, where `$DATADIR` is the local pathname of the directory where the server stores all the documents. The version *VerNum* of a document *docname* is stored as the file `$DATADIR/docname/VerNum.hash` on the local hard disk, where *hash* is the hash of the contents of the version *VerNum*. Thus, to arrive at this sorted list, a server first reads the list of subdirectories in `$DATADIR` to determine the list of documents, and then reads the highest filename in each subdirectory to arrive at the highest version for a document. The function call `scandir()` available in Linux and

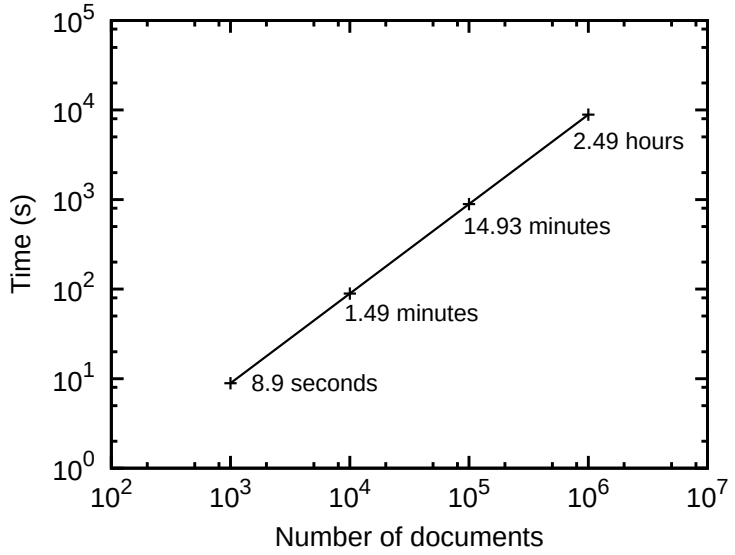

Figure 12: Time taken to compute and check the hash of the highest version of a document vs. the number of documents

BSD is used to read the entries in a directory in sorted order.

We ran experiments for 100, 1000, and 10,000 documents stored in the system, with three versions stored for each document. For all these, it took approximately 3 seconds to complete PIVR_Step1.

PIVR_Step1 must complete as quickly as possible so that the remainder of the epoch is used for running PIVR_Step2, which is a resource intensive process. The three seconds taken by PIVR_Step1 is negligible compared to the epoch length, which we have assumed to be one day. PIVR_Step1 is thus scalable with large numbers of stored documents.

6.5.2 Scalability of PIVR_Step2

PIVR_Step2 performs two resource-intensive functions - (F1) compute the hash of the highest version of each document stored on disk, and (F2) repair corrupted files by reading them off other servers. We present results for these below.

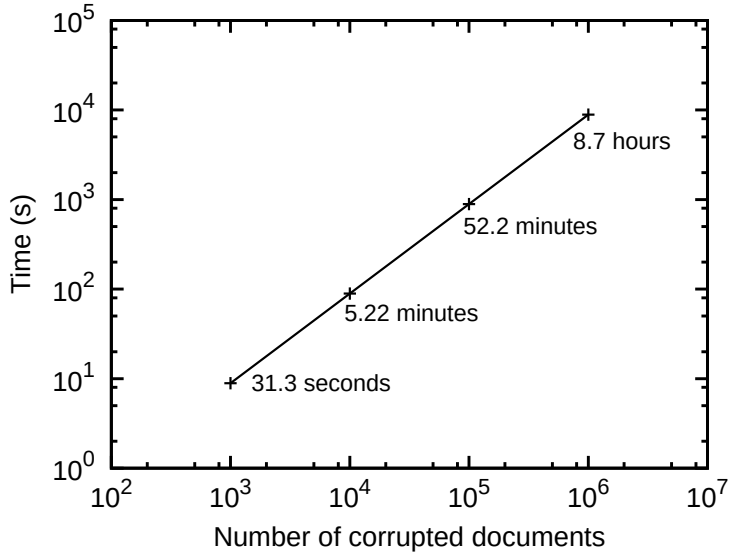

Figure 13: Time taken to repair corrupted documents vs. the number of corrupted documents

We found that the rate at which a server computes the hash (using SHA-160) of a 1 MB file to be 112.36 files per second. We measured this by taking the average over the hash computations over 1000 different 1 MB files with random content. Figure 12 shows the time taken to compute and check the hashes of the highest version of the documents vs. the number of documents.

If the hash check for a document version fails, then it must be read off a server that has the correct contents of the document version. In our experiment, we corrupted the contents for 1000 documents at a server, and measured the time the server took to repair its corrupted data by reading these documents off another server. The servers were not processing any reads and writes during these experiments. Figure 13 shows the time taken for a server to repair its corrupted data vs. the number of corrupted documents.

From Figures 12 and 13, we see that for a storage system consisting of 100,000 or less documents (100 GB or less), the time taken to execute PIVR may be over an

hour in the worst case, where all 100,000 documents are corrupted at a server. This is not significant compared to the assumed epoch length of one day. However, for a storage system consisting of a million documents (over 1 TB), the time taken for executing PIVR may be long (over eleven hours) in the worst case. PIVR_Step2 can thus scale to over a terabyte of data storage in a system where reads and writes are infrequent.

6.6 Conclusions

This chapter addresses the problem of maintaining the integrity of the stored data under the mobile adversary model. In every epoch, an adversary can compromise some server and arbitrarily corrupt the data stored at the server. Protocol PIVR described in the chapter is run by the servers at the beginning of every epoch to repair any data corruptions from earlier epochs.

The data stored at the servers can be classified into two types - metadata and userdata. Protocol PIVR is a two step process. In the first step, servers synchronize with each other on the metadata. In the second step, the servers check the locally-stored userdata against the metadata and repair corrupted userdata. We require that the metadata for an object contain the hash (checksum) of the contents of the object.

Experimental analysis shows that the first step in Protocol PIVR completes in only three seconds. The experimental analysis also shows that the second step in Protocol PIVR is a time consuming operation. When the system is not processing reads and writes, the second step takes over an hour when 100,000 or less 1 MB-sized documents are stored in the system, but over eleven hours to complete when a million 1 MB-sized documents are stored in the system.

CHAPTER VII

BYZANTINE FAULT DETECTION IN QUORUM SYSTEMS

7.1 Introduction

A necessary component in any secure system design is intrusion detection. The mobile adversary model assumes that an adversary moves from one storage server to another, and that only up to a threshold of servers can be compromised in an epoch. To maintain these assumptions, vulnerabilities must be identified and fixed. The fact that vulnerabilities exist in the system become evident when intrusions are detected.

In this chapter, we restrict to server intrusions resulting in Byzantine faults. A fault-free server can detect if it was Byzantine-faulty in an earlier epoch if it finds any corruptions in its stored data during the Periodic Integrity Verification and Repair process. Corruptions of the stored data is one manifestation of a Byzantine fault. Another type of manifestation of a Byzantine fault is arbitrary and harmful behavior during protocol executions. Such behavior can be detected during the protocol execution by other servers or clients.

We study the problem of detecting Byzantine server faults during read operations in the context of Byzantine quorum protocols, or Byzantine quorum systems. In Byzantine quorum systems, writes and reads take place to only subsets or quorums

of servers, with the requirement that the read and write quorums overlap. Since writes take place to only a quorum of servers instead of all the servers, a fault-free server could return an outdated response during reads. Thus, it is not trivial to detect Byzantine server faults during reads in a Byzantine quorum system.

The next section describes the related works in this area. The rest of the chapter gives an overview of quorum systems, the fault detection algorithm, and experimental results.

7.2 Related Work

The primary work in fault detection for Byzantine quorum systems is [10]. One of the two algorithms proposed in this paper, called Diagnosis using Justifying Sets, cannot identify individual faulty servers and is useful only in detecting if the actual number of faulty servers is close to the preset fault threshold b. It is assumed that faulty servers almost always return incorrect responses. The other algorithm, called the Write Marker Protocol, can identify faulty servers in a single read operation provided it is guaranteed that no read will be concurrent with a write. The Write Marker protocol incurs an overhead of n bits per data object, where n is the total number of servers in the system. This overhead can be quite significant if a large number of data objects are stored.

The overhead incurred in the proposed fault detection algorithm described in this chapter is the tracking of read responses of every server in the store by the proxy servers. This storage overhead does not depend on the number of data objects in the system. For each server in the store, proxy servers maintain a chronological list of read operations in which the server participated. Each entry in the list contains the quorum timestamp associated with the data object that was the result of the corresponding read operation, and a Boolean value indicating if the server returned

the correct response. Since quorum parameters are assumed not to change often, two bytes should be sufficient to hold each entry in the list. Assuming a maximum of 1000 read operations are monitored for each server, a storage space of 2000 bytes will be required. For a store of size 70 servers, each proxy will need approximately 140 KB of main memory space, which is independent of the number of data objects in the system and can be easily accomodated with typical memory configurations in today's server class machines.

As mentioned earlier, our approach can tolerate a limited amount of concurrency between read and write operations, while the Write Marker approach cannot. In Section 7.6, simulation results demonstrating this capability are presented.

7.3 System Model and Architecture

Figure 14 shows the system architecture of the Byzantine quorum system. It consists of the storage servers, clients, and a *diagnosis server*. Each data object is replicated at a subset, or quorum, of servers. The storage servers monitor each other during protocol execution and report their findings to the diagnosis server, which makes the final decision on whether a server is Byzantine-faulty. It is assumed that only up to a threshold b of the storage servers can be Byzantine-faulty. The network communication links are assumed to be asynchronous but reliable.

We assume that clients are not Byzantine faulty. We focus only on the detection of Byzantine behavior of the storage servers. In practice, the servers can employ intrusion detection mechanisms and detect and prevent some faulty client behavior such as writing to a partial quorum. Access control mechanisms also limit the number of data objects that can be corrupted by Byzantine-faulty clients. The PIVR protocol can be used to detect and repair corrupted writes by a Byzantine-faulty client. Faulty clients can have some impact on the accuracy of our fault detection technique. This

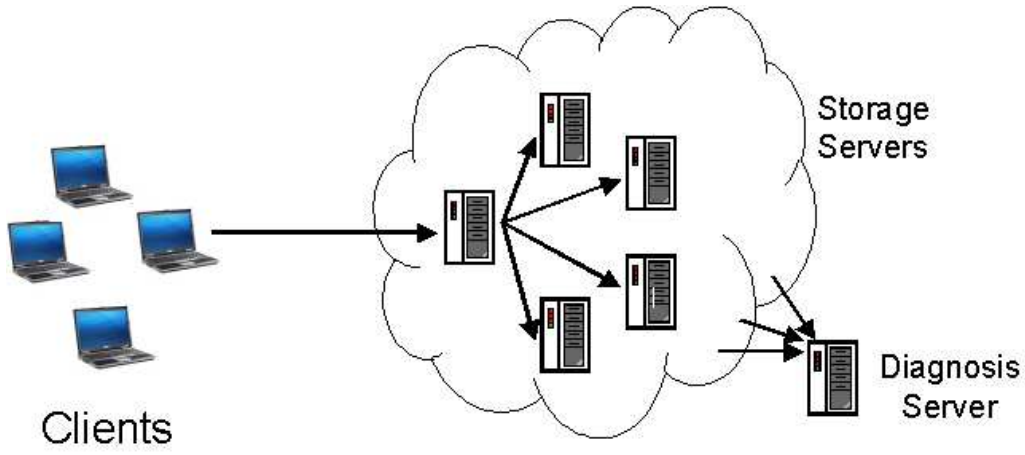

Figure 14: Byzantine quorum system architecture

issue is discussed when describing the fault detection algorithm.

The diagnosis server, which is responsible for collecting fault information from the proxy servers and making the final decision on whether a server is Byzantine-faulty, is assumed to be fault-free. The diagnosis server does not, in general, accept connections from the outside world, and communicates with only the storage servers to receive some fault information. Furthermore, the diagnosis server runs a single, very simple application. These elements should make it feasible to guarantee the security and protection of the diagnosis server. Alternatively, the diagnosis server could be implemented as a Byzantine fault-tolerant state machine.

We assume authentication, authorization, and key management services for the system, and digital certificates from a certificate authority bind all parties to their public keys. All parties use cryptographic methods to protect the confidentiality and integrity of their communication.

7.4 Byzantine Quorum Protocols

Quorum systems are replication-based techniques. Clients read and write *data objects*. Reads and writes take place to a quorum or subset of the entire set of servers in the system, and sufficient overlap between any two quorums guarantees that the latest value will be read. Timestamps are used to distinguish values written in different write operations to the same data object. In Byzantine quorum systems [44], the minimum overlap between any two quorums is increased sufficiently in order to accomodate Byzantine failures.

The fault detection algorithm is developed in the context of *b-masking quorum systems* [44], but other types of quorum systems such as *dissemination* [44] quorum systems can also be used.

A general description of b-masking quorum systems is as follows: To read a data object, the client queries a read quorum Q_{r} of servers for the timestamp and the associated value for the data object. Among the responses from a read quorum, the client chooses the data object that has the highest timestamp and seconded by at least $b + 1$ servers, where b denotes the Byzantine-fault threshold. To write a new value for a data object, the client first executes a read on the data object's timestamp, increments it to a higher value, and then writes the new value and the new timestamp to a write quorum Q_{w} of servers. The read and write quorums must overlap in at least $2b + 1$ servers. Since at least $b + 1$ non-faulty servers are in $Q_{\mathrm{r}} \cap Q_{\mathrm{w}}$, a read on a data object that is not concurrent with any writes to the same data object will return the value written in the last write.

We modify the above protocol to use proxy, or gateway, servers. To read and write data objects, a client picks a storage server at random as a proxy, or gateway, server. The client sends its read or write request to the chosen proxy server, and

the proxy server forwards the request to the quorum of servers specified by the client in its request. Server responses are also channeled through the same proxy server back to the client. Proxy servers can thus monitor the responses of other servers, enabling them to detect Byzantine-faulty behavior. Cryptographic methods are used to protect messages from being tampered by faulty proxy servers.

7.5 Fault Detection Algorithm

A novel fault detection algorithm that identifies Byzantine-faulty servers thus enabling their removal is described in this section. The Byzantine quorum system we consider uses *proxy* servers, which accept requests from clients, contacts a quorum of servers, and channels the responses back to the clients. The proxy servers can thus monitor server responses, especially during read operations.

In the proposed algorithm, the proxy servers monitor server responses during reads and, over time, are able to determine if other servers are Byzantine-faulty with a specified false-alarm probability. Proxy servers communicate their findings to the diagnosis server, which makes the final decision on whether a server is indeed Byzantine-faulty. Since faulty proxy servers may report non-faulty servers as faulty, the diagnosis server employs a voting mechanism to decide if a server is indeed faulty. Hence, the fault detection algorithm uses a two-tiered approach: the proxy-server algorithm and the diagnosis-server algorithm.

7.5.1 The Fault Detection Algorithm at the Proxy Server

To better illustrate the algorithm, we make some simplifying assumptions. The fault detection algorithm is run periodically on batches of reads and, in between two executions of the algorithm, the following assumptions hold:

1. No server becomes faulty.
2. There are no concurrent reads and writes.
3. A quorum of size q is chosen randomly from all possible quorums of size q.

These assumptions will be relaxed in Section 7.5.3.

During a data read operation, the proxy server is aware of the responses returned by the servers in the read quorum. Hence, it can determine the result of the read as would be chosen by the client, which will be termed here as the "correct response." In a read operation, some servers will be unable to give the correct response because they were not part of the write quorum of the last completed write to the data object. Hence, to be able to distinguish faulty servers from non-faulty servers using this approach, statistical analysis is used over a sequence of read operations.

When a non-faulty server is part of a read operation on a data object, the probability that it will return a correct response is the probability that it belonged to the write quorum of the last completed write on the same data object, which is $|Q_{\mathrm{w}}|/n$. If read operations to r objects are monitored, then the probability that a fault-free server returns u correct responses is given by

$$\binom{r}{u} \left(\frac{|Q_{\mathrm{w}}|}{n}\right)^{u} \left(1 - \frac{|Q_{\mathrm{w}}|}{n}\right)^{r-u} \tag{5}$$

A well known statistical technique called *Hypothesis Testing* is used to determine if a server is faulty. More specifically, the null hypothesis, H_0, is defined as a server being non-faulty, and for a fixed false-alarm probability the null hypothesis H_0 is either rejected or accepted. The false-alarm probability is fixed at 0.05, meaning the probability that a non-faulty server is found faulty by a fault-free proxy server is 0.05. From the false alarm probability, a threshold value for u, denoted by u_{th}, is

determined, and if a server returns u_{th} or fewer correct responses in r read operations, then the null hypothesis H_0 is rejected and the server is said to be faulty. u_{th} is the maximum value of u such that the following inequality is satisfied:

$$\sum_{i=0}^{u} \binom{r}{i} \left(\frac{|Q_{\text{w}}|}{n}\right)^{i} \left(1 - \frac{|Q_{\text{w}}|}{n}\right)^{r-i} \leq 0.05 \tag{6}$$

Since Byzantine faults are considered, faulty servers could try to avoid detection by sporadically giving incorrect responses. Since the number of faulty servers is never greater than b, incorrect responses by faulty servers will not affect the result of a read operation when the read is not concurrent with a write to the same data object. Note that faulty servers do, however, have the incentive to respond incorrectly as soon as they are compromised because, under Byzantine quorum system operation, they can force arbitrarily old values to be read when a read operation is concurrent with a write to the same data object.

Let p_{ic} denote the probability with which a faulty server returns an incorrect response when it has the correct value for the data object being read. p_{ic} is a probability representative of the faulty server's behavior over the sequence of r read operations being monitored and cannot be estimated or observed. An ideal fault detection algorithm would detect a faulty server for all p_{ic}, $0 < p_{\text{ic}} \leq 1$.

The probability that a faulty server returns a correct response during a read operation is $\frac{|Q_{\text{w}}|}{n}(1 - p_{\text{ic}})$ and the probability that an incorrect response is returned is $1 - \frac{|Q_{\text{w}}|}{n} + \left(\frac{|Q_{\text{w}}|}{n}\right) p_{\text{ic}}$. Since a server is said to be faulty if it returns u_{th} or fewer correct responses in r read operations, the probability that a faulty server will be detected as faulty in r read operations is

$$\sum_{i=0}^{u_{\text{th}}} \binom{r}{i} \left(\frac{|Q_{\text{w}}|}{n}(1 - p_{\text{ic}})\right)^{i} \left(1 - \frac{|Q_{\text{w}}|}{n} + \frac{|Q_{\text{w}}|}{n} p_{\text{ic}}\right)^{r-i} \tag{7}$$

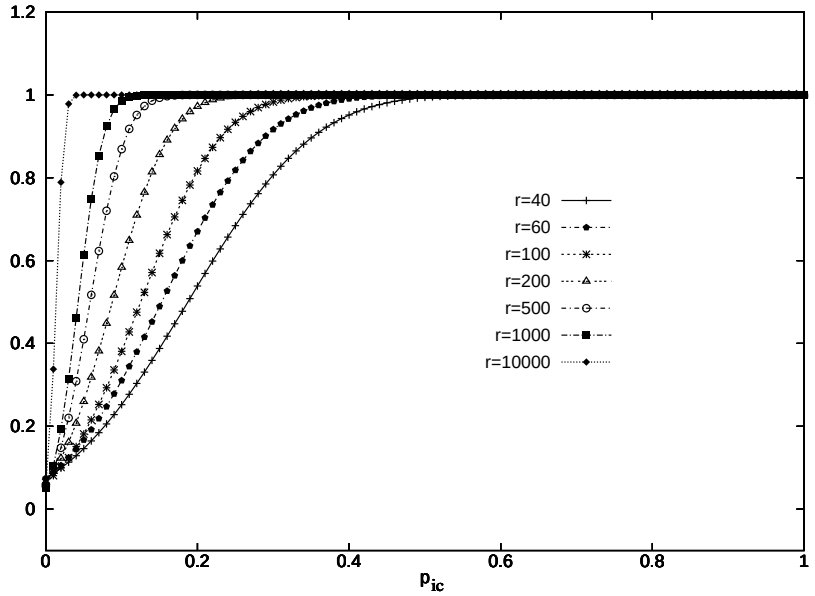

Figure 15: Probability that a faulty server is detected vs p_{ic} for several r

For a store consisting of $n = 70$ servers and with a write quorum size $|Q_{\text{w}}| = 42$ servers, Figure 15 shows the probability that a faulty server is detected for various values of p_{ic} and r with the false alarm probability kept equal to 0.05. Even for r as low as 100 read operations, the probability that a faulty server is detected is one for a wide range of p_{ic}. The probability that a faulty server is detected improves as the number of monitored read operations increases.

It is interesting to note that a faulty server has to behave at a very small p_{ic} to avoid detection. Thus, the algorithm forces faulty servers to behave almost as though they were correct in order to avoid detection.

Proxy servers run the above described algorithm periodically for each server in the store. At the end of one such period for a monitored server, the proxy server determines if the monitored server is faulty with the specified false alarm probability and informs the diagnosis server of the result.

7.5.2 The Fault Detection Algorithm at the Diagnosis Server

The diagnosis server maintains for each server in the store a list of servers that found this server to be faulty. This list may increase or decrease with time. If at any point, a certain minimum number of servers, denoted by m, claim that some server is faulty, then that server is diagnosed as faulty. The value of m can be chosen to achieve a false alarm probability lower than the false alarm probability used by the proxy fault detection algorithm. The choice of m is also influenced by the number of concurrent reads and writes in the system, which are a potential source for false alarms.

For example, if the final desired false alarm probability is 10^{-4}, then let m'' be the smallest m' such that the following inequality is satisfied:

$$\left(\begin{array}{c} n - b_{\max} \\ m' \end{array} \right) (0.05)^{m'} (1 - 0.05)^{n - b_{\max} - m'} \leq 10^{-4} \tag{8}$$

where 0.05 is the false alarm probability used in the proxy-server fault detection algorithm. $n - b_{\max}$ and not n is used in the above inequality because, in the worst case, $b_{\max}$ faulty servers could try to vote a non-faulty server out of the system. Hence, m given by $m'' + b_{\max}$ guarantees a final false alarm probability of at most $\leq 10^{-4}$.

A faulty server could defeat the above algorithm by returning incorrect responses only when a particular set of fewer than m servers are proxy servers, and behaving correctly when other servers are proxy servers. Since the diagnosis server will then not be able to gather sufficient (m) votes to identify the faulty server, the faulty server will go undetected. Hence, we make the following assumption.

Assumption 4: The behavior of faulty servers is independent of the identity of the proxy servers.

The above assumption can be relaxed by having proxy servers drop servers they have found to be faulty from quorums specified by a client. The client will then not be

able to get a quorum of responses and will generate requests to other servers, ensuring that the read or write completes. Another technique is to have a proxy server tunnel the client's requests through other proxy servers before the request reaches a quorum of servers. Thus, faulty servers will not be able to single out specific proxy servers. Evaluating these ideas more thoroughly is an area for future work.

7.5.3 Relaxing Assumptions 1 - 3

Assumption 1 states that no servers become faulty during the execution of the fault detection algorithm. If a server becomes faulty during the r read operations that were monitored, the average p_{ic} over the r read operations will be lower than the actual p_{ic} that is representative of the faulty server's behavior. Hence, if a server became faulty recently, the effective p_{ic} over the r read operations monitored could be small enough so that it goes undetected. Then, the faulty server will be detected in the next round of r read operations provided it continues to perform with a p_{ic} in the detectable range. Small values of r are particularly useful in detecting servers that became faulty recently. Simulation results in Section 7.6 and experimental results in Section 7.7 show that even with r as low as 100 read operations, faulty servers with small p_{ic} are detected.

If Assumption 2 is relaxed, then concurrent reads and writes can occur. It could then be possible that a wrong "correct" response is determined during a read operation, thereby counting fault-free servers as faulty. This would increase the actual false alarm probability to be higher than the target value used in Equation 8. This can be dealt with simply by lowering the target false alarm probability to counter the impact of concurrency.

Assumption 3 is used when choosing write quorums in the analysis in Section 7.5.1. If the quorum selection strategy is defined such that all the possible quorums of size

q are not equally likely to be chosen during write operations, then the fault detection algorithm can be suitably modified. Assumption 3 could be *violated* when Byzantine-faulty clients do not adhere to the chosen strategy. For example, Byzantine-faulty clients can deliberately exclude a particular non-faulty server in their write operations. Although we do not consider faulty clients in this chapter, we note that it is possible to have proxy servers monitor clients' quorum selections to detect this type of behavior.

7.6 Simulation Analysis

7.6.1 Fault detection in a Reconfigurable Byzantine Quorum System

Reconfigurable Byzantine quorum systems were introduced in [39]. These quorum systems are b-masking quorum systems where the fault threshold b can be varied dynamically and faulty servers can be removed while the system is in operation without affecting the correctness of the read-write protocols. A reconfigurable Byzantine quorum system which used the proposed fault detection algorithm to identify Byzantine faulty servers was simulated. A system size of 70 storage servers was assumed, with the fault threshold b ranging within $[B_{\min} = 1, B_{\max} = 8]$. The fault threshold b is updated as soon as a server becomes faulty or when a faulty server is identified and removed from the system. The simulation was done using discrete-event simulation techniques. In the simulations, the Byzantine-fault threshold b is maintained as the actual number of faulty servers in the system plus one to model a safety margin that should be built into the b estimation.

The time at which a server becomes faulty is computed for each server using an exponential distribution with mean $MTBF\sqrt{\frac{B_{\min}+B_{\max}}{2(\#faulty\ servers)}}$, where $MTBF$ is the Mean Time Between Failures for a server and $\#faulty\ servers$ is the total number of servers that became faulty since simulation start. Thus, the effective $MTBF$ decreases as the number of faulty servers increase. This is intended to model scenarios

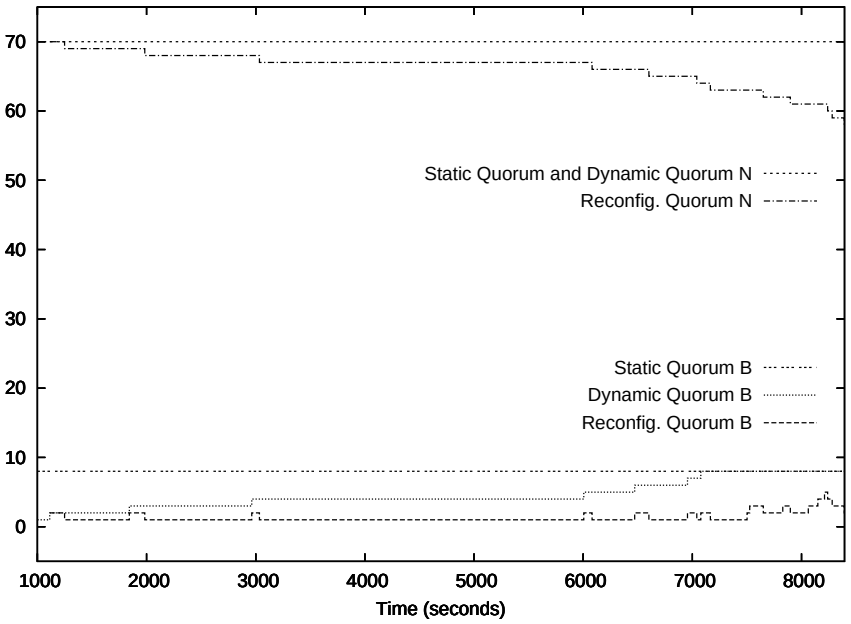

Figure 16: Variation of system size N and the fault threshold b in a reconfigurable Byzantine quorum system.

such as virus propagation and progressive attacks. The value of $MTBF$ used in the simulations is 2 days. At system start, there are no faults in the system. Hence, at start, b is equal to 1. When a server becomes faulty, the p_{ic} value that will govern its subsequent behavior is chosen uniformly over $(0, 1]$.

The time taken for a message to traverse a communication link is a minimum message travel time (t_1) + a random amount of time exponentially distributed with mean t_2. In the simulations, t_1 is 1 second and t_2 is 4 seconds. Read and write requests for each data object are modeled as Poisson processes. For each object, the mean interarrival time for reads is 80 seconds and the mean for writes is 150 seconds, thus producing an average read/write ratio of 15 : 8.

Figure 16 shows the variation in the system size N and the fault threshold b during the length of one simulation run. Since the fault detection algorithm is incorporated into the simulation of reconfigurable quorum systems, faulty servers are removed some time after the fault event thus maintaining the fault threshold b fairly constant.

The points where the fault threshold decreases are the times when a faulty server is identified and removed from the system by the diagnosis server. The false alarm probability is set to 0.05 in the proxy-node fault detection algorithm and 10^{-20} in the diagnosis-node fault detection algorithm. For a system size of 70 servers and with $B_{\max} = 8$, at least 38 servers must notify the presence of the faulty server to the diagnosis node before the faulty server is removed from the system. Proxy servers run the fault detection algorithm on other servers every $r = 100$ read operations. There were no incorrect diagnosis even though the measured concurrency between read and write operations was 32.63%.

In the depicted simulation, our fault detection algorithm identified all but one failure and the average latency in detecting faults was found to be 164 seconds. The fault event at time 8062.04 seconds remained undetected because the p_{ic} value governing the faulty server's behavior was a very small 0.0125. On the other hand, a faulty server with a p_{ic} as low as 0.0775 was detected, albeit with a high latency of 714 seconds. In general, it was observed that faulty servers that behave with a small p_{ic} require a long time to be detected. From Figure 15, we see that, with $r = 100$, it is remarkable that a faulty server with a p_{ic} equal to 0.0775 was detected, and a faulty server with $p_{ic} = 0.0125$ going undetected is not unexpected. The smallest latency in diagnosing a faulty server was found to be 68.99 seconds, corresponding to a p_{ic} value of 0.807.

Reconfigurable Byzantine quorum systems impose certain lower bounds on the number of storage servers that must be present in the system. The simulation is run until no more servers can be removed and the fault threshold has reached its maximum value. Thus, the system lifetime for reconfigurable quorum systems was found to be 9885.12 seconds. Note that the MTBF value of 2 days used in the simulation is quite low so as to generate a large number of failures in a short period. Therefore, the

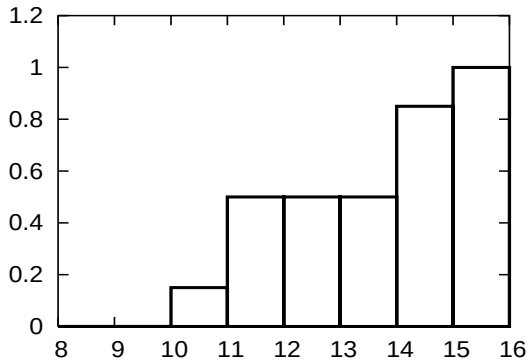

Figure 17: Fraction of Simulation Runs with Incorrect Diagnosis vs. Percentage of Reads Concurrent with Writes

system lifetime found in the simulation is much lower than they would be in practice.

7.6.2 Concurrency Analysis

Figure 17 shows the performance of the fault detection algorithm in the presence of concurrent reads and writes for the same system parameters used in Section 7.6.1. In Section 7.6.1, the read and write requests were generated by separate Poisson processes. To do concurrency analysis, finer control over concurrency is needed. We achieved this by having, for each data object, a write request generated periodically at the same time as read requests. For example, if 50% concurrency is desired, then a write request is issued with every other read request. In each concurrency interval shown in the figure, 20 simulations were run and the number of simulations that had any incorrect diagnoses were noted. This is shown as a fraction of the total number of simulations run for that particular concurrency interval.

Generation of write requests at exactly the same time as read requests produces maximum overlap between them. It can, therefore, be considered as the worst-case concurrency scenario for a particular concurrency rate. From Figure 17, we find that no incorrect diagnoses were produced in 20 simulation runs at a concurrency rate in the range $9\% - 10\%$, while the number of runs with incorrect diagnoses rose rapidly

as the concurrency rate increased beyond this point. In Section 7.6.1, there were no incorrect diagnoses reported despite a concurrency rate of more than 32%. In those simulations, however, read and write operations began at random times and concurrency happened naturally as a result. When read and write operations overlap only partially, incorrect diagnosis becomes less likely than in the maximum overlap case. If the approximate concurrency rate in the system can be predicted, it can be factored into the diagnosis threshold calculation and incorrect diagnosis can be eliminated even for higher concurrency rates. Verification and quantification of this idea is an area for future work.

7.7 Evaluation in the AgileFS prototype

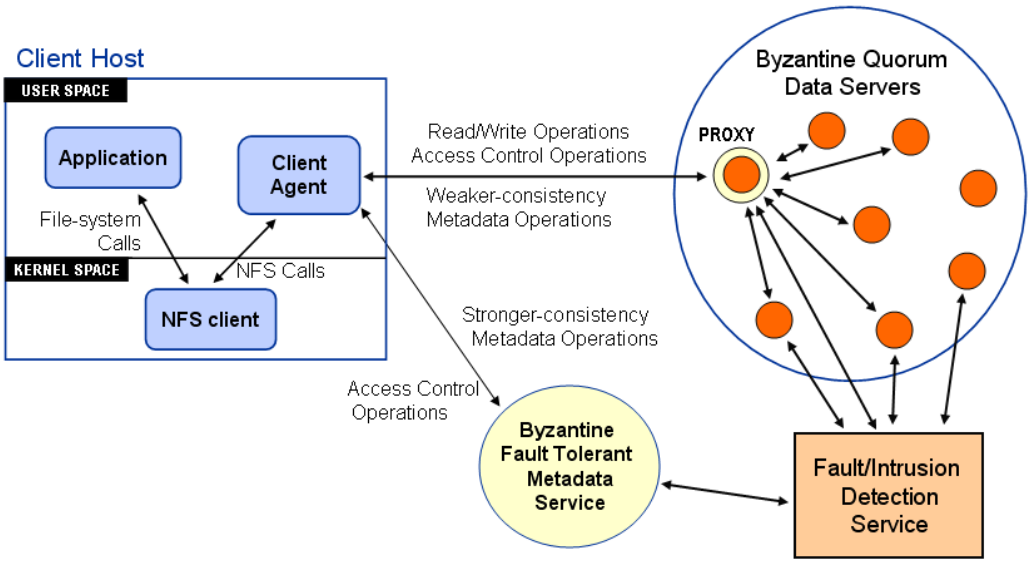

Figure 18: Schematic overview of the AgileFS distributed filesystem.

The proposed Byzantine-fault detection algorithm was implemented in the AgileFS [38] distributed filesystem prototype. Figure 18 depicts a schematic overview of the prototype. The prototype was implemented on the Emulab cluster [2]. The machine testbed consisted of Intel PIII 600 MHz processors with 256 MB RAM and

13 GB 7200 RPM IDE hard disks, and Intel PIII 850 MHz processors with 512 MB RAM and 40 GB 7200 RPM IDE hard disks. All machines ran RedHat Linux 7.3. A block size of 8 KB was used. The network is a 100 Mbps switched Ethernet.

A set of *data servers* store file blocks and attribute information using reconfigurable Byzantine quorum systems, while the metadata is implemented using a state machine approach at a separate set of *metadata servers*. The fault detection algorithm at the proxy server was run at the data servers. The fault / intrusion detection service was implemented as a single instance of a diagnosis server. Communication between the diagnosis server and the data servers was over UDP.

Figure 19 shows the variation of the number of data servers in the system and the fault threshold in one sample run where data servers became faulty over time. The variations in N and b for reconfigurable quorum systems are compared against the case where the fault threshold is preset to a conservatively high value (static Byzantine quorums [44]), and the case where the fault threshold is increased to mask new faults but no faulty servers are removed (dynamic Byzantine quorums [9]).

The experiment was run with 35 data servers, fed with a continuous stream of read operations. The fault threshold b was allowed to vary between $B_{\min} = 1$ and $B_{\max} = 5$ servers. At start, there were no faulty data servers in the system. Servers became faulty at random times, but the likelihood of additional servers being faulty was increased each time a new server became faulty. The points where the fault threshold b decreases are the times when a faulty server is identified and removed by the diagnosis server. The experiment demonstrates that reconfigurable Byzantine quorum systems can tolerate more faults and have a longer system lifetime that dynamic and static Byzantine quorum systems. Since the fault threshold b can increase only up to $B_{\max} = 5$ servers, only five faults can be injected into dynamic and static quorum systems. By contrast, a total of 11 faults were introduced in the case of reconfigurable quorums due

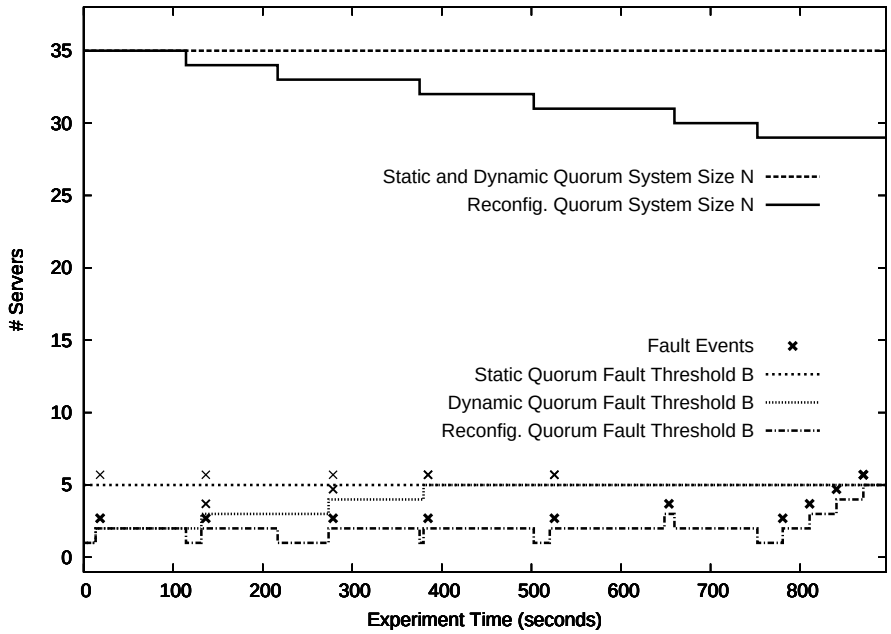

Figure 19: The variation of the number of data servers (N) and the fault threshold (b) when data servers become faulty over time.

to the timely detection and removal of faulty servers, which keeps the fault threshold b small. The proposed fault detection algorithm thus plays a key role in lengthening the ssytem lifetime of reconfigurable Byzantine quorum systems.

7.8 Conclusions

A Byzantine-fault / intrusion detection service is an essential component in any secure system design. An intrusion indicates vulnerabilities exist in the system, which must be removed. The system is thus made more secure, and assumptions on the number of compromised servers in an epoch as in the mobile adversary model are more likely to hold.

This chapter presents a Byzantine-fault detection mechanism that detects Byzantine behavior of servers during reads in a Byzantine quorum system. In a fully replicated system, when reading a certain data object, if a server does not give a value

identical to other servers then that server has to be Byzantine-faulty. In quorum systems, however, data objects are replicated only at a quorum of servers instead of full replication. Thus, even fault-free servers may return incorrect values during reads. The algorithm proposed in this chapter can detect Byzantine faults in quorum systems.

The proposed algorithm monitors server responses over sequences of read operations. The algorithm has the property that a Byzantine-faulty server should behave as a fault-free server for most of the reads to avoid being detected. By arbitrarily increasing the length of the sequence of read operations being monitored, a Byzantine-faulty server could be made to behave almost identically to a fault-free server to avoid being detected.

Reads concurrent with writes in quorum systems may return arbitrarily old values. Thus, even fault-free servers could be diagnosed as Byzantine faulty if there is a sufficient level of read-write concurrency. Simulation results show that the proposed algorithm can tolerate high levels of concurrency - over 32%. The algorithm was also implemented and evaluated in the AgileFS distributed filesystem prototype.

CHAPTER VIII

PROTOTYPE IMPLEMENTATION AND EVALUATION

This chapter describes a prototype implementation of a proactively-secure data storage service. The prototype is designed to store *documents*. Users can upload a new version of a document, or download the latest version of a document, or delete documents. A typical deployment scenario of this prototype is for the storage of sensitive documents in a corporate LAN. Users can access these documents over the company intranet, or externally over a wide area network. The software for this prototype can be downloaded from http://www.arunsubbiah.com/publications/proactive_store.tgz.

Byzantine and leakage-only faults are considered in the prototype. Data is stored using the encrypt-and-replicate storage model. We use the prototype implementation to determine the amount of data that can be stored in a practical proactively-secure distributed storage system, and the impact of running Protocol PIVR on the upload and download throughput and latency.

8.1 Prototype Description

8.1.1 Overview

Figure 20 gives an overview of the document storage system. The functions of each component are described below:

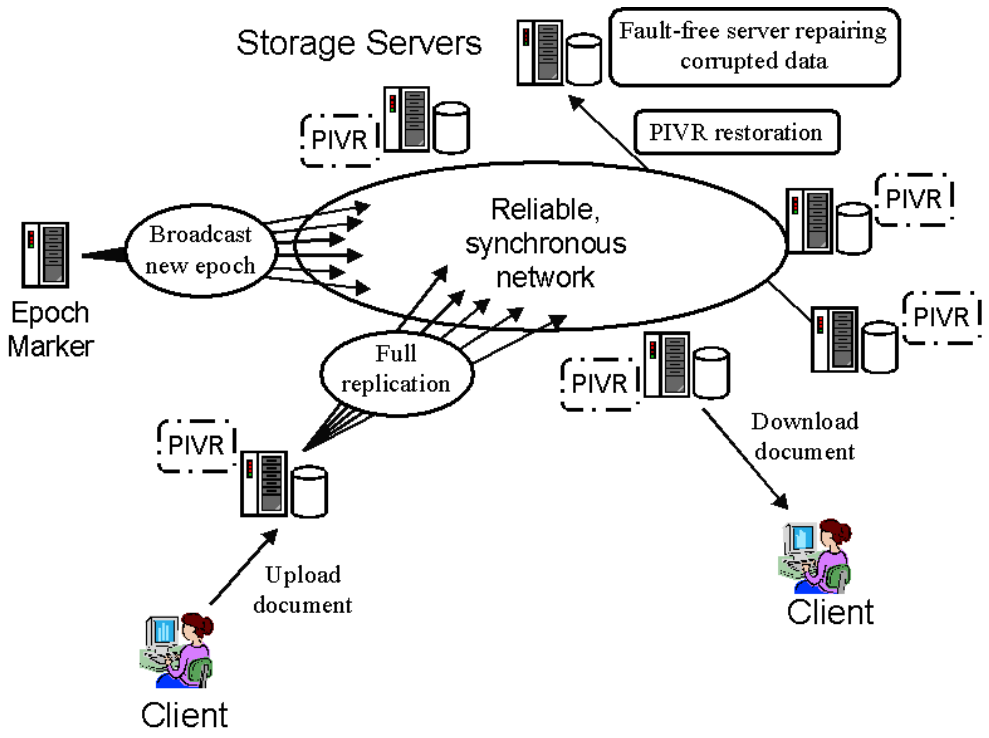

Figure 20: Overview of the document repository prototype.

Clients: The clients are software agents acting on behalf of the users. The client program takes as input a list of documents and whether the documents are to be uploaded (writes), or downloaded (reads), or deleted. When a document *docname* is uploaded, the file is uploaded as a new version of document *docname*. When a document is deleted, all versions of the document stored thus far are no longer available for download.

Epoch Marker: The Epoch Marker is always fault-free, and divides time into *epochs*. At the start of a new epoch, it sends a <New_Epoch, *epoch number*> message to all the servers over the reliable synchronous network.

Storage Servers: The storage servers store documents, and execute Protocol PIVR (Chapter 6) periodically to maintain the integrity of the stored documents under

the mobile adversary.

We assume a flat namespace, and that the document names are globally unique. All versions of a document *docname* are stored in the directory `$DATADIR/docname`, where `$DATADIR` is the local pathname of the directory where the server stores all the documents. The version number *VerNum* is of the format Serial_Num||User_ID||hash, where the User ID is the ID of the user who wrote that version, the hash is in hex format and is computed over the contents of that version, and || denotes concatenation. The version *VerNum* of a document *docname* is stored as the file `$DATADIR/docname/VerNum` on the local hard disk.

The Epoch Marker is a process running on a separate server machine that wakes up periodically and notifies all the servers the start of a new epoch. A server, upon learning the start of a new epoch, starts running Protocol PIVR. As part of PIVR_Step2, the servers keep the latest versions of the documents and remove all prior versions.

All client-server and server-server communication is using TCP. The Epoch Marker notifies the start of a new epoch to all the servers using UDP. Reliability over UDP is implemented using retransmissions-upon-timeout until acknowledgement messages are received from the servers.

Clients perform uploads, downloads, and deletes by issuing requests over the client-server network to a server, chosen at random, called the proxy server. The proxy server performs fault-tolerant replication of a document during uploads. All messages are signed using Message Authentication Codes (MACs) for verifying authenticity and integrity; for example, an ACK sent by a server to a client via a proxy server contains a MAC signed by the server that the client can verify. Clients establish symmetric session keys with all the servers prior to performing uploads, downloads, and deletes.

To write a new version of document docname*, or to delete document* docname*, a client does*:

1. Connect to a server chosen at random, called the *proxy server*.
2. Query for the highest version number of the document *docname*.
   Version numbers are of the format Serial_Number||User_ID||hash.
3. Wait for responses from $(2b + 1)$ servers.
   Responses from servers are received via the proxy server.
4. Determine the response with the highest version number returned by at least $(b + 1)$ servers.
   Denote this response as *Resp*.
   If no such response exists, go to Step 1.
5. Choose a serial number greater than the serial number in *Resp*.
   Denote this serial number by $\text{SerialNum}_{\text{new}}$.
   If *Resp* is DOC_NOT_EXIST, then set $\text{SerialNum}_{\text{new}}$ to 0.
6. To write a new version, send request
   <`WRITE`, *docname*, $\text{SerialNum}_{\text{new}}$, document_contents>
   to the proxy server.
   To delete a document, send request
   <`DELETE`, *docname*, $\text{SerialNum}_{\text{new}}$, NULL>
   to the proxy server.
7. Wait for responses from at least $(N - b)$ servers.
8. If less than $(b + 1)$ servers report ACK, goto (1), choosing a different proxy.
9. End.

Figure 21: Protocol followed by clients to write new versions of documents and to delete documents.

8.1.2 Write Protocol

Figures 21-23 give the protocols followed by the clients and the servers to write new versions of a document and to delete documents. The clients communicate with a server chosen at random, called a *proxy server*. The proxy server forwards the request to other storage servers and acts on behalf of the client. Deletes are treated as a special form of writes. Clients connect to proxy servers over the reliable asynchronous network, while all inter-server communication is over the reliable synchronous network.

Each server (including the proxy server), upon receiving a `WRITE` or `DELETE` request

Upon receiving a write or delete request from a client, the proxy server executes:

1. Forward request to all the servers over the reliable and synchronous network.
2. Execute a distributed consensus protocol involving all servers on whether
   the client's request for document *docname* with version number *VerNum*
   was received.
3. If Step 2 returns *success*,
   return ACK to the client.
   Store document *docname* with version number *VerNum*.
   else
   return NACK to the client.
   Discard document *docname* with version number *VerNum*.
4. Receive responses (ACK or NACK) from all other servers,
   and forward them to the client.
5. End.

Figure 22: Protocol followed by proxy servers to process client write and delete requests.

from a client, participates in a Byzantine fault-tolerant distributed consensus protocol on the validity of the request. If all servers that are not Byzantine faulty agree that the client's request is valid, then the write or delete request is accepted, otherwise the request is rejected. Depending on whether the request is accepted or rejected, the servers return an ACK or a NACK response to the client via the proxy server. The distributed consensus protocol ensures that at least $(N - 2b)$ servers agree on the validity of the request. This safeguards against Byzantine-faulty proxy servers that may not forward the write or delete request to all the servers. In addition, a Byzantine-faulty proxy server may tamper the client's write or delete request before forwarding the request to other servers. To prevent this, clients compute a Message Authentication Code (MAC) for each server and include it in their write and delete requests. A storage server, upon receiving a write or delete request from a proxy server, will verify the correctness of the MAC to determine if the request is valid.

In the prototype implementation, we use the Byzantine fault tolerant distributed

Upon receiving a write or delete request from a client via the proxy server, a storage server executes:

1. Execute a distributed consensus protocol involving all servers on whether the client's request for document *docname* with version number *VerNum* was received.
2. If Step 1 returns *success*,
   return ACK to the client via the proxy server.
   Store document *docname* with version number *VerNum*.
   else
   return NACK to the client via the proxy server.
   Discard document *docname* with version number *VerNum*.
3. End.

Figure 23: Protocol followed by storage servers to process client write and delete requests.

consensus algorithm given in [31]. The distributed consensus algorithm requires at least $(4b+1)$ servers and completes after $(b+1)$ rounds of communication, with each round consisting of two phases.

A proxy server that is Byzantine-faulty may not forward the client's request to all the servers, leading to the client incurring a "soft" timeout waiting for ACKs in Step 7 of Figure 21. The client can then retry by sending the `WRITE` or `DELETE` request to a different proxy server. A Byzantine-faulty proxy server must also be unable to spoof client requests. To address this, we have the clients establish symmetric keys with each server which are valid for the duration of a session. These symmetric keys are used to protect the integrity of the messages. The message sender computes a Message Authentication Code (MAC) over the message and appends it to the message. The message receiver will accept the message only if the MAC is found to be valid. For example, a client will compute, for each server, a MAC over the write request, and then send the MACs as part of the write request to the proxy server. The responses from the servers (ACK or NACK) received by the client via the proxy server are also

protected from tampering using MACs.

Clients embed their view of the current *epoch number* in all their requests. A server accepts a request for processing only if its view of the epoch number matches the number contained in the client's request.

8.1.3 Handling Deletes

Deletes are treated as *writes* involving a special "Delete" file. The protocols followed are the same as in writes, and are described in Section 8.1.2. This section describes the local processing at a storage server when storing the "delete" file in Step 2 of Figure 23 and Step 3 of Figure 22.

The version number *VerNum* of a document *docname* is of the format Serial_Num||User_ID||hash, where *hash* is the hash of the contents of the document. The document version is stored as the file `$DATADIR/docname/VerNum` on the local hard disk. For a "delete" file, a zero byte file is created with the *hash* part of *VerNum* string replaced with the string "DEL".

As a result, the document *docname* and all its versions are still stored on the hard disk. The read protocol in the following section takes deleted documents into account; clients can detect if a document is deleted and will be unable to read prior versions.

The cleanup is performed during PIVR_Step2. After the servers have agreed on a common list of document names and the highest version numbers, if the highest version number of a document is of the form Serial_Num||User_ID||DEL, then the document and all its prior versions are erased from the hard disk.

To read document docname*, a client does*:

1. Connect to a server chosen at random, called the *proxy server*.
2. Query for the highest version number of the document *docname*.
   Version numbers are of the format Serial_Number||User_ID||hash.
3. Wait for responses from $(2b + 1)$ servers.
   Responses from servers are received via the proxy server.
4. Determine the response with the highest version number
   that is returned by at least $(b + 1)$ servers.
   If no such response exists, go to Step 1.
   Denote this response by *Resp*.
5. If the version number in *Resp* is of the format Serial_Number||User_ID||DEL,
   or if *Resp* is DOC_NOT_EXIST,
   then return DOC_NOT_EXIST, End.
6. Choose a server for downloading the document.
   Send request <`READ`, *docname*, VerNum> to the server, and receive document.
   VerNum is the version number specified in *Resp*.
   If the server does not have the specified document version, repeat,
   choosing a different server.
   If no server has the specified document version, go to Step 1.
7. Check if hash of received document matches the hash contained in *VerNum*.
   If mismatch, go to Step 6, choosing a different server.
8. End.

Figure 24: Protocol followed by clients to read a document.

8.1.4 Read Protocol

Figure 24 gives the protocol followed by the clients to read a document. During writes and deletes, the use of a distributed consensus protocol ensures that all non-Byzantine-faulty servers agree upon the validity of the write and store the received version (or delete the document). Thus, to read the latest version of a document, the client needs to query only $(2b + 1)$ servers. Since a maximum of b servers can be Byzantine-faulty, the version returned by at least $(b + 1)$ servers is definitely from at least one fault-free server.

In Step 6 of Figure 24, the client reads the document off a server that reported the latest version. If the proxy server to which the client is already connected to reported

the latest version, then the document is downloaded from the proxy server. This is almost always the case, since document versions are replicated at all servers.

8.2 Integrating Protocol PIVR to achieve Proactive Security

The servers execute Protocol PIVR (Chapter 6) in the beginning of every epoch to maintain the integrity of the stored documents in the mobile adversary model. Protocol PIVR imposes the following requirements on the write protocol:

PIVR_WProp1: Writes update the metadata at all the servers that are not Byzantine faulty. The metadata that is updated includes the document name and the checksum of the contents of the document version included in the write request.

PIVR_WProp2: Writes store the contents of the new version at at least $(b + 1)$ servers that are not Byzantine faulty.

PIVR_WProp3: Servers that are not Byzantine faulty in the previous and current epochs can consistently identify the userdata stored till the last epoch.

PIVR_WProp4: Each server that is not Byzantine faulty can safely conclude, at some point in the current epoch, that it has received all metadata updates due to writes from the previous epoch.

Satisfying Requirement PIVR_WProp1: The distributed consensus protocol, which is used during writes and deletes, has the following property:

Agreement: Two non-Byzantine-faulty servers cannot decide on different values. Each server participates in the distributed consensus protocol with an initial value v_0. When a server finishes participating in the distributed consensus protocol, it has decided upon an outcome with can take one of different values, including v_0. The

Agreement property says that all non-Byzantine-faulty servers agree on the same outcome.

In our prototype, the values are either 0 or 1. The value v_0 is set to 0. A server that participates in the consensus protocol with an initial value of 0 indicates that the server either did not receive the client's write or delete request, or it received the write or delete request and found it to be invalid; an initial value of 1 indicates that the server received the client's write or delete request and found it to be valid. If the servers *decide* on a value of 1, then they *accept* the write or delete request and store it permanently. The version is then available for reads. If the servers decide on a value of 0, then they discard the write or delete request.

The distributed consensus protocol run is identified by the metadata in the write or delete request, which is the document name and the version number. The version number contains the hash of the contents of the document version. The Agreement property of the distributed consensus protocol ensures that all non-faulty servers are thus updated with the metadata during writes, satisfying Requirement PIVR_WProp1.

Satisfying Requirement PIVR_WProp2: During writes, a distributed consensus protocol on the validity of the write request is run amongst the servers. If a server receives an invalid write request, it discards it along with the document contents included in the request. Each server participates in the distributed consensus protocol run associated with the write request with an *initial value*, which could be 0 or 1 - depending on whether the server received the client's request and found it valid. The Agreement Property of the distributed consensus protocol guarantees that all non-Byzantine-faulty servers decide on the *same outcome*: either the write is valid or not. Thus, a non-Byzantine-faulty server could start with an initial value of 0 and decide on a value of 1.

The distributed consensus protocol of [31] has the property that if at least $\left(\left\lfloor \frac{N}{2} \right\rfloor + b + 1\right)$ non-Byzantine-faulty servers start with an initial value v, then the decided value will be v. Thus, the decided outcome can be 1 even when up to $\left(\left\lfloor \frac{N}{2} \right\rfloor + b\right)$ non-Byzantine-faulty servers start with an initial value of 0. That is, a successful write will store the contents of the document version at at least $\left(\left\lceil \frac{N}{2} \right\rceil - 2b\right)$ non-Byzantine-faulty servers. Thus, to meet Requirement PIVR_WProp2,

$$\left(\left\lceil \frac{N}{2} \right\rceil - 2b\right) \geq b + 1$$

giving the following lower bound on the number of storage servers N:

$$N \geq 6b + 1 \tag{9}$$

A server can start with an initial value of 0 in the distributed consensus protocol due to the following reasons: 1) A Byzantine client can force a server to start with an initial value of 0 by generating an incorrect MAC for the server, which is included in the write request. And 2) a Byzantine proxy server can arbitrarily tamper with a client's request before forwarding it to a server to cause a mismatch with the client-generated MAC for the server.

The distributed consensus is on the document name and version number, which contains the hash of the document contents. Thus, all non-Byzantine-faulty servers will have the document metadata (the document name and its version number) that was included in the client's write request, but up to $\left(\left\lfloor \frac{N}{2} \right\rfloor + b\right)$ of them may not have the document contents.

Satisfying Requirement PIVR_WProp3: We have already shown that the protocols satisfy Requirement PIVR_WProp1, which ensures that all non-faulty servers have their metadata updated by writes. Requirement PIVR_WProp3 requires the

servers to have a consistent technique to classify these updates by epoch numbers. We achieve this as follows: All client write requests contain the client's view of the epoch number. At the start of a new epoch, the Epoch Marker sends a <New_Epoch, *epoch number*> message to all the servers over the reliable synchronous network. A server, upon receiving this message, stops accepting new writes and aborts concurrent uploads from the clients and proxy servers, and participates only in any on-going consensus protocol runs. Clients can immediately reconnect and retry the write, but the write request will now have the new epoch number embedded in it.

Satisfying Requirement PIVR_WProp4: Metadata updates at all non-faulty servers is achieved using the distributed consensus protocol run during writes. The servers must have a way of determining when the distributed consensus protocol runs due to writes initiated in the previous epoch have gone to completion. The servers can then start running Protocol PIVR, as new writes and distributed consensus protocol runs will only affect the metadata and the userdata stored in the current epoch.

The solution implemented in the prototype is as follows: When a server A receives a <New_Epoch, *epoch number*> message from the Epoch Marker, it is already running a set S of distributed consensus protocol runs due to writes in the previous epoch. It continues to *actively* participate in these distributed consensus protocol runs, that is, it sends and receives consensus protocol messages. The server *passively* participates in new distributed consensus protocol runs due to writes in the previous epoch, that is, it only receives consensus protocol messages, but does not send any. The server classifies these new distributed consensus protocol runs into a set S'. Needless to say, the sets S and S' do not overlap. When Server A completes executing a distributed consensus protocol in either of these sets, it removes it from the set.

Because Server A passively participates in the distributed consensus protocol runs in Set S', other servers will timeout waiting for Server A's message and continue with the distributed consensus protocol runs. To avoid incurring timeouts due to passive participations, when Server A finds its Set S empty, it sends a special message to all the other servers indicating that it will be participating passively in all current and future distributed consensus protocol runs due to writes from the previous epoch.

A peer server will interpret this special message as Server A always sending a value of 0 to it in all its distributed consensus protocol runs. The distributed consensus protocol we use has the property that if at least $\left(\left\lfloor \frac{N}{2} \right\rfloor + b + 1\right)$ non-faulty servers start with an initial value of 0, then the decided value will be 0. A decided value of 0 implies that the write must be discarded, thus leaving the metadata and the userdata of the stored documents unaffected.

Thus, when Server A determines that at at least $\left(\left\lfloor \frac{N}{2} \right\rfloor + 2b\right)$ servers (an additional b servers because the server does not have a reliable technique to detect Byzantine-faulty peer servers) will be passively participating in its distributed consensus protocol runs it can safely conclude that all distributed consensus protocol runs in its Set S' will end with a decided value of 0 and the corresponding write requests can be discarded.

Requirement PIVR_WProp4 is thus met, as each non-faulty server can safely conclude, at some point in the current epoch, that it has received all metadata updates due to writes from the previous epoch.

Notice that distinct sets of b servers can be Byzantine faulty in the previous and current epochs, which should impose the requirement that the distributed consensus protocol runs that straddle epoch boundaries tolerate $2b$ Byzantine faulty servers. We circumvent this requirement by making the following assumption:

Assumption: Servers that are Byzantine-faulty at some point between the time instant the Epoch Marker determines the start of a new epoch and the time instant the last non-faulty server has finished executing all distributed consensus protocol runs due to writes from the previous epoch are considered to be Byzantine-faulty throughout the previous and current epochs, and thus count towards the Byzantine fault threshold in both epochs.

8.3 Experimental Evaluation

8.3.1 Experimental Setup

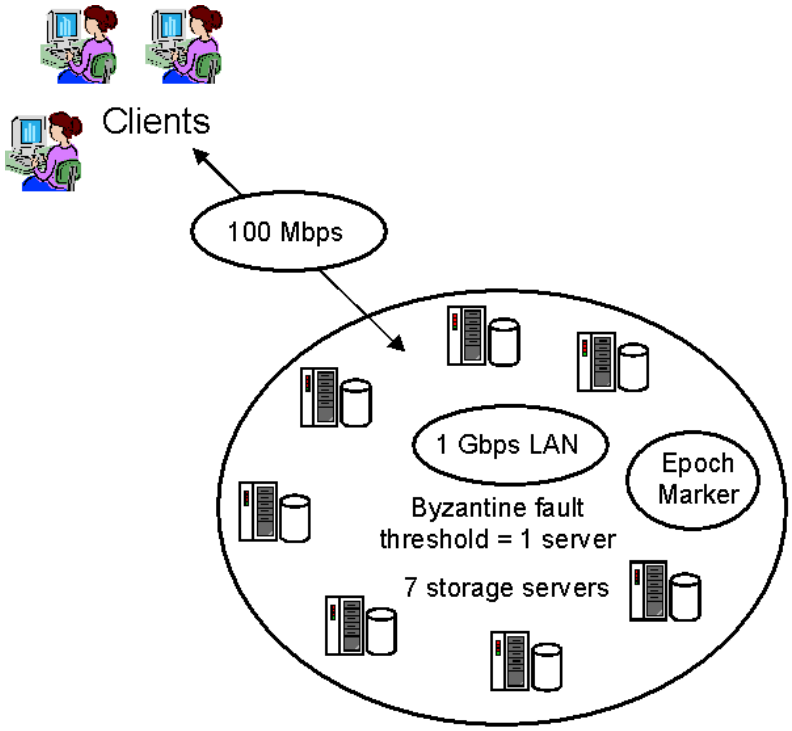

Figure 25: Experimental setup of the document repository prototype.

The prototype implementation was run on the Emulab cluster (http://www.emulab.net). The setup is shown in Figure 25. Both clients and servers

are powered by a 3 GHz 64-bit Xeon processor with 2 GB RAM and 10K RPM 146 GB hard disk. Clients connect to the servers over a 100 Mbps switched Ethernet LAN, in which large latencies for emulating a wide area network can be achieved by using nodes that delay-and-forward network packets. All inter-server communication is over a 1 Gbps switched Ethernet LAN. We consider only Byzantine faults among the servers, and set the Byzantine fault threshold b in every epoch to be one. The number of servers is set to the minimum required, which is $(6b+1) = 7$ servers. Each client process performs downloads (reads) and uploads (writes) in the ratio 4:1. The size of the documents is set to 1 MB. Each server stores 500 documents.

8.3.2 Effect of Integrity Verification and Restoration on Throughput

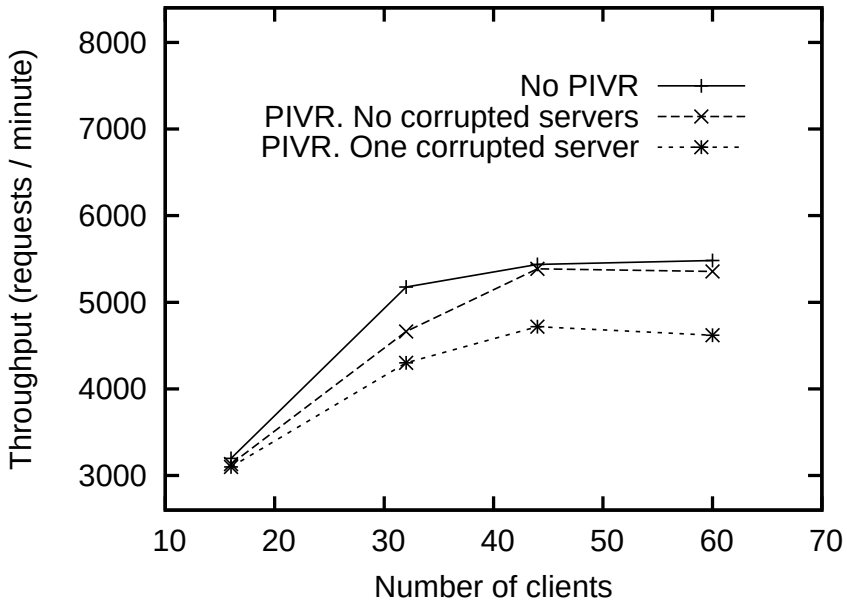

Figure 26: R/W Throughput Vs. number of clients with and without PIVR.

In this experiment, we measure the effect of the PIVR process on the upload / download throughput of the storage service. Figure 26 shows the read/write (or download/upload) throughput as a function of the number of client processes. Each

client process performs downloads (reads) and uploads (writes) in the ratio 4:1. We found that running four client processes per client machine saturates the 100 Mbps network interface at the client machine.

When none of the servers are storing any corrupted data, then as part of the PIVR process, all the servers will be checking the integrity of the stored data and will find no corrupted data; so there will not be any repair process running in the system. The maximum throughput of the system is achieved with 44 client processes, and the throughput is 5386 requests / minute. This throughput is less than 1% lower than the throughput for 44 client processes when none of the servers are running Protocol PIVR. However, more than 44 clients are needed to achieve the maximum throughput when none of the servers are running Protocol PIVR.

When one of the servers is storing corrupted data, then that server detects the corruption during the integrity verification process and repairs itself by reading the correct contents from another server. The maximum throughput in this case is achieved with 44 client processes, and the throughput is 4719 requests / minute. This throughput is 13.22% lower than the throughput for 44 client processes when none of the servers are running Protocol PIVR. The large drop in throughput during the repair is because the corrupted server repairs itself by reading the contents from another server over the 1 Gbps Ethernet LAN, while the clients connect to the servers over the 100 Mbps LAN. Thus, two out of the seven servers have part of their resources devoted to the repair process and do not serve client requests at maximum capacity.

Thus, when none of the servers are storing corrupted data, Protocol PIVR has little impact on the throughput of the system. However, when one of the servers is storing corrupted data and is repairing itself, the throughput drops noticeably by over 13%.

8.3.3 Effect of PIVR Integrity Verification on Read-Write Latency

This experiment examines the effect of Protocol PIVR on the upload and download latencies when none of the servers are storing corrupted data. Table 18 gives the measurements.

Table 18: Effect of PIVR integrity verification on upload and download latencies.

	Without PIVR	With PIVR	% increase
Download	97.81 ms	109.19 ms	11.63%
Upload	169.16 ms	239.67 ms	41.68%

Running Protocol PIVR causes the download latency to increase by over 11% and the upload latency to increase by over 41%. Thus, Protocol PIVR has a noticeable impact on the upload and download latencies. The difference in impact on the two latencies is because, in case of downloads, the file is downloaded from only one server, while for uploads the file is copied to all servers and then the servers run a distributed consensus protocol on the write request. The upload process is a longer operation consisting of multiple steps involving all the servers.

The impact of PIVR on the latencies can be reduced by designing the server hardware appropriately. For example, each server could consist of a high-performance disk connected to two machines over fiber channel; one of the machines services client requests, while the other runs Protocol PIVR.

8.3.4 PIVR Integrity Restoration Rate Vs. #Clients

This experiment examines the effect of increasing the number of clients on the integrity restoration rate at a server storing corrupted data. Figure 27 shows the rate at which the server which is storing corrupted data detects the corruption and repairs it as the number of clients accessing the servers increases. The integrity restoration rate at the server drops off almost linearly as the number of clients increases. When

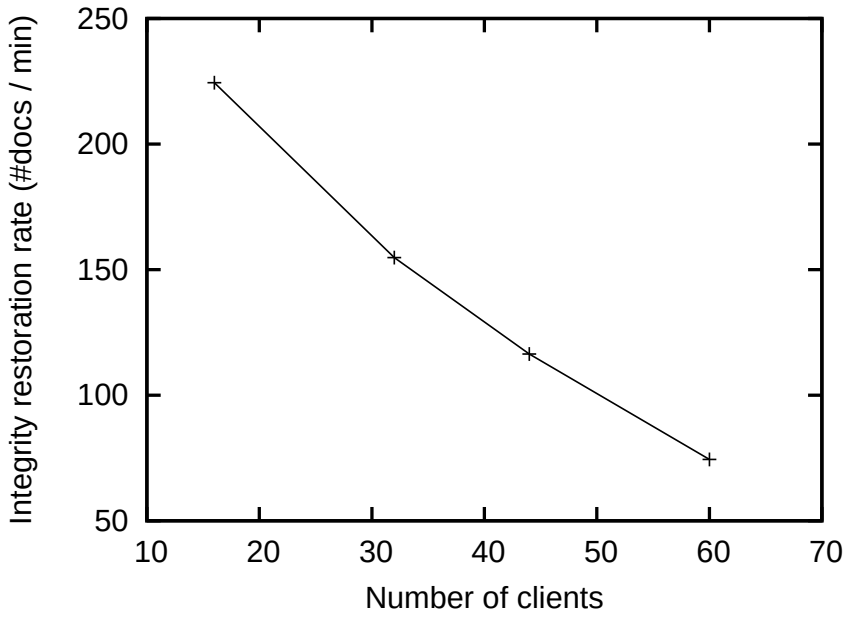

Figure 27: Rate at which corrupted documents are repaired Vs. number of clients.

the number of clients is 60, the restoration rate is 74.4 documents per minute. If the epoch length is set to one day, then the server will be able to restore 107 GB of stored data in an epoch. The integrity restoration component of Protocol PIVR thus scales to a few hundred gigabytes of stored data.

8.3.5 Effect of Integrity Verification Process Priority on Integrity Verification Rate

This experiment examines the impact of the scheduling priority of the PIVR process on the rate at which the integrity of the documents are checked. Figure 28 shows the rate at which servers verify the integrity of their stored documents as a function of the number of client processes for two process scheduling priorities of the integrity verification process. Each client process performs downloads (reads) and uploads (writes) in the ratio 4:1. We run four client processes per client machine.

The priority with which an operating system runs a process is called the "niceness"

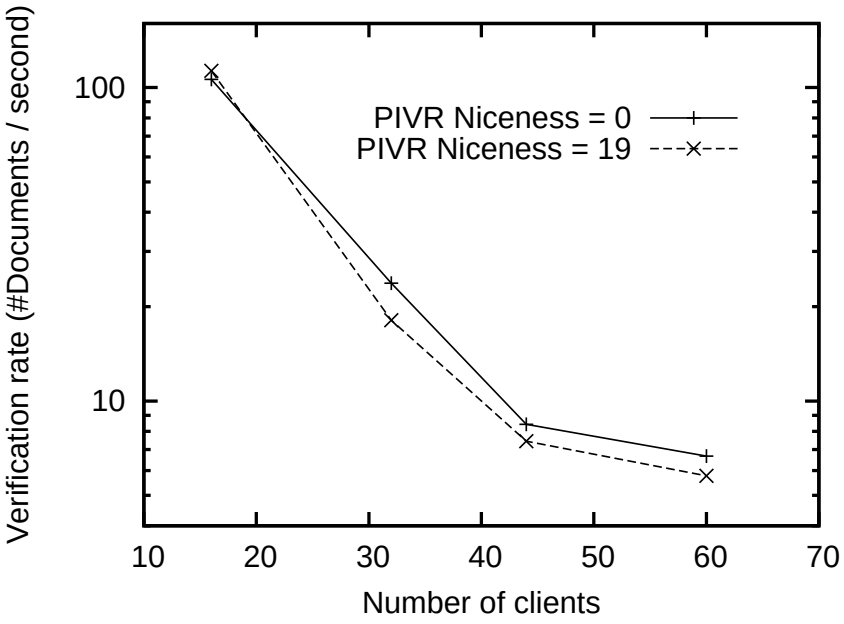

Figure 28: Rate at which document integrity is verified Vs. number of clients for different values of process scheduling priority.

value of the process. The default niceness value of a process is zero. Increasing the niceness value of the process implies the process is run with a lower priority by the operating system. The maximum niceness value on Linux is 19.

The figure shows that lowering the priority of the integrity verification process (increasing the niceness value) decreases the integrity verification rate by 11.63% when there are 44 client processes in the system. This may be acceptable depending on how much data is stored in the system. If the epoch length is set to one day, then with an integrity verification rate of 10 1-MB sized documents / second, the integrity of 864 GB of data can be verified in an epoch. This also shows that the integrity verification component of Protocol PIVR scales to hundreds of GB's of data.

8.3.6 Wide Area Network Performance

This section describes read-write performance and document integrity verification rates when clients connect to the servers over a wide area network. Emulation of a wide area network on the Emulab testbed is achieved by using "delay" nodes in the network path between the clients and the servers. The delay nodes work at the IP layer of the network protocol stack. The clients and the servers are set to communicate over 100 Mbps network links with 12 ms round trip times.

8.3.6.1 Effect of PIVR Integrity Verification on Read-Write Latency

This experiment examines the effect of Protocol PIVR on the upload and download latencies of a 1 MB document when none of the servers are storing corrupted data. Table 19 gives the measurements.

Table 19: Effect of PIVR integrity verification on WAN upload and download latencies.

	Without PIVR	With PIVR	% increase
Download	0.463 s	0.51 s	10.15%
Upload	0.53 s	0.585 s	10.38%

Compared to the LAN setup (Table 18), the latencies have increased by 2.4 to 5 times. Running Protocol PIVR causes the download and upload latencies to increase by a little over 10%. Compared to Table 18, the % increase in the upload and download latencies due to Protocol PIVR are almost equal, because the client-server network latency is the largest contributing factor to the upload and download latencies.

8.3.6.2 Effect of PIVR Integrity Verification on Throughput

In this experiment, we measure the effect of the PIVR integrity verification process on the upload / download throughput of the storage service. None of the servers are

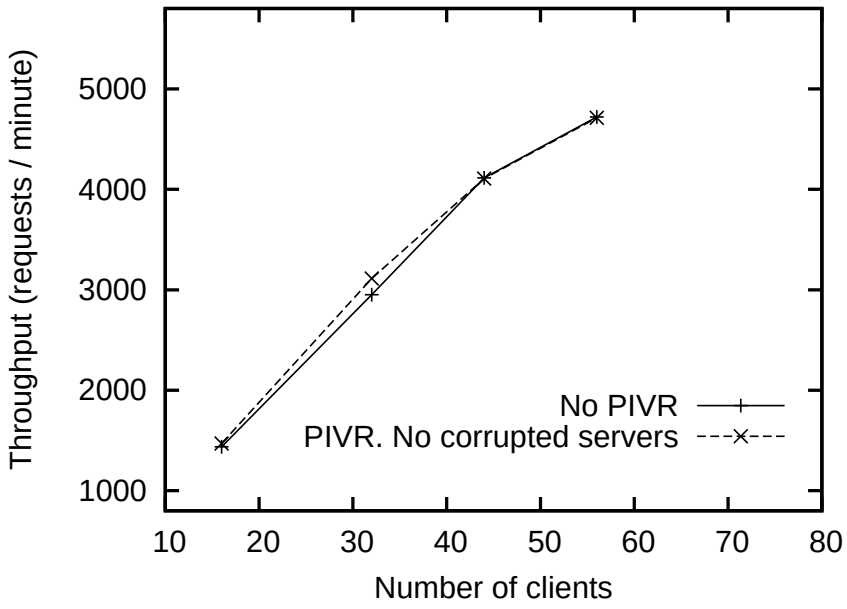

Figure 29: WAN R/W Throughput Vs. number of clients with and without PIVR.

storing any corrupted data. Figure 29 shows the read/write (or download/upload) throughput as a function of the number of client processes. Each client process performs downloads (reads) and uploads (writes) of 1 MB-sized documents in the ratio 4:1. We found that the maximum throughput that can be extracted using only one client machine is by running four client processes per client machine.

Figure 29 shows that running Protocol PIVR has almost no effect on the throughputs. Compared to the LAN setup (Figure 26), the maximum throughput saturation is not reached even with 56 client processes. This is expected, because the packets from each client reach the servers at a slower rate, thus requiring a greater number of clients to saturate the servers with requests.

8.3.6.3 Effect of Number of Clients on PIVR Integrity Verification Rate

This experiment examines the impact of the number of clients on the rate at which the integrity of the documents are checked. Figure 30 shows the rate at which servers

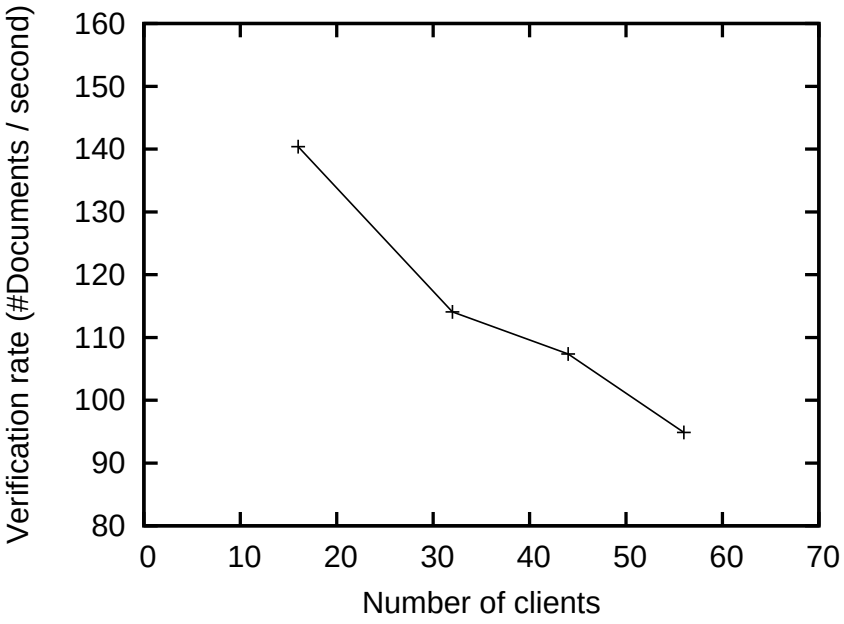

Figure 30: Rate at which document integrity is verified Vs. number of clients (WAN).

verify the integrity of their stored documents as a function of the number of client processes when the clients connect to the servers over the wide area network. None of the servers are storing any corrupted data.

Comparing Figure 30 with the setup where the clients connect to the servers over a LAN (Figure 28), two observations can be made: 1) The integrity verification rate drops off linearly with number of clients in the WAN setup, while it drops off exponentially in the LAN setup; and 2) The integrity verification rate in the WAN setup is greater (even for 16 client processes) than in the LAN setup.

Both these observations are due to the increased round trip time in the network links connecting the clients with the servers. During uploads and downloads, the transfers are done in blocks of 16 KB using TCP. With higher network latencies, the server processes spend more time waiting for blocks to arrive (or to send, during

downloads). More system resources are thus available for the PIVR integrity verification process, which explains Observation 2. Conversely, the lower the network latencies are, the lower the PIVR integrity verification rate will be. Observation 1 shows that the client-server network latency influences greatly the PIVR integrity verification rate, with the influence becoming more pronounced as the number of clients is increased.

8.4 Conclusions

This chapter describes a prototype implementation of a proactively-secure distributed data storage service, and an experimental evaluation demonstrating its feasibility in practical settings. The prototype implements a document storage service, where users can upload new versions of a document, download the latest version of a document, and delete documents. Protocols realizing this service were developed and implemented as part of the prototype implementation.

Experimental results for the LAN setup (client-server network is a 100 Mbps LAN) show that the integrity verification component of Protocol PIVR has little impact on the system throughput; has a noticeable impact on the upload and download latencies; and can secure 100s of GB of data even when run under a low scheduling priority. The experimental results also show that the integrity restoration rate has a noticeable impact on the system throughput, and can scale to securing a few hundred GB's of stored data. Thus, the proactively-secure document repository as a whole can scale to a few hundred GB's of stored data.

Experiments for the WAN setup (client-server network consists of 100 Mbps network links with 12 ms RTT) were done and compared to the results for the LAN setup. The integrity verification component of Protocol PIVR has almost negligible impact on the system throughput. The differences in the experimental results of the

WAN and LAN setups are: 1) The integrity verification component of Protocol PIVR causes almost the same % degradation in the upload latency as in the download latency for the WAN setup, while in the LAN setup the % degradation in the upload latency is almost four times that of the download latency; 2) The integrity verification rate in the WAN setup is greater (even for 16 client processes) than in the LAN setup; and 3) The integrity verificate rate drops off linearly with number of clients in the WAN setup, while it drops off exponentially in the LAN setup. Difference #3 (linear drop off for WAN vs. exponential drop off for LAN) shows that the client-server network latency influences greatly the PIVR integrity verification rate, with the influence becoming more pronounced as the number of clients is increased.

CHAPTER IX

CONCLUSIONS AND FUTURE WORK

9.1 Dissertation Summary

Fault tolerant and secure distributed data storage systems typically require that only up to a threshold of storage nodes can ever be compromised or fail. In proactively-secure systems, this requirement is modified to hold only in a time interval (also called epoch), resulting in increased security. An attacker or adversary could compromise distinct sets of nodes in any two time intervals. Proactively-secure systems thus require all nodes to "refresh" themselves periodically to a clean state to maintain the availability, integrity, and confidentiality properties of the data storage service.

This dissertation investigates the design of a proactively-secure distributed data storage system. The protocols that are run by the servers to refresh themselves periodically must scale with the large amounts of stored data. We consider two storage models - secret sharing and encrypt-and-replicate. For data stored using secret sharing, the confidentiality of the encoding must be maintained using a share renewal process, while for data stored using encryption, we assume that the users manage the encryption keys securely. The contributions made in this research are:

- For storing data, we consider, in addition to the standard encrypt-and-replicate storage model, data storage using perfect secret sharing schemes. We show

that standard secret sharing schemes have high computation overheads and are impractical for storing large amounts of data. A new technique called the GridSharing framework is proposed that uses a combination of XOR secret sharing and replication for storing data efficiently. The number of rows in the GridSharing framework is a configurable parameter that can be varied to achieve a tradeoff in performance metrics.

- We give a share renewal algorithm in the context of the GridSharing framework for maintaining the confidentiality of the stored data under the mobile adversary model. We experimentally show that the algorithm can secure several hundred GBs of data.

- We give distributed protocols run periodically by the servers for maintaining the integrity of replicated data under the mobile adversary model. We experimentally show that these protocols scale to several 100 GBs of stored data.

- We design a proactively-secure document repository, where users can upload a new version of a document, download the latest version of a document, and delete documents. The read-write protocols are specified. The protocol run periodically for maintaining the integrity under the mobile adversary model is integrated into the prototype, and concurrency and timing issues at epoch boundaries are addressed. The result is an integrated system that maintains correctness in the presence of concurrent executions of the read-write protocols and the protocol for maintaining integrity in the mobile adversary model.

- The proactively-secure document repository is implemented and evaluated on the Emulab cluster (http://www.emulab.net). The experimental evaluation shows that several 100 GBs of data can be proactively-secured.

- A necessary component in any secure system is fault and intrusion detection. We give a novel Byzantine-fault detection algorithm for quorum systems, and experimentally evaluate it using simulations and by deploying it in the AgileFS distributed file system. Concurrent reads and writes are a source for false alarms in Byzantine fault detection algorithms. We show that the proposed fault detection algorithm does not produce incorrect diagnosis even when the read-write concurrency rate is as high as 32%. In addition, an interesting property of the fault detection algorithm is that a Byzantine-faulty server has to mimic closely a fault-free server to avoid being detected.

9.2 Future Work

The research and results in this dissertation open up some new research directions, listed below:

- We have shown that proactively-secure systems can scale to 100s GB of data, as measured on the Emulab testbed. The periodic refresh mechanisms provided in this dissertation can be applied in other systems to make them proactively-secure. Such mechanisms should be run anyway as a good system administration practice.

- The GridSharing framework provides an efficient technique for storing data using perfect secret sharing schemes. If generic (n, n) linear secret sharing schemes are used instead of XOR secret sharing, then computations over stored data can be performed by performing distributed computations over their shares. The distributed computations need not be made "verifiable" (in the sense of verifiable secret sharing) because of the use of replication-with-voting. The distributed computations can thus be efficient and practical.

REFERENCES

[1] "The Agile Store Project." http://www.ece.gatech.edu/research/labs/agile_store. Last accessed: May 2007.

[2] "Emulab." http://www.emulab.net. Last accessed: June 2007.

[3] "The MIRACL software library." http://indigo.ie/ mscott/. Last accessed: December 2005.

[4] "Tripwire." http://sourceforge.net/projects/tripwire/. Last accessed: June 2007.

[5] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J., "Lazy verification in fault-tolerant distributed storage systems," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 2005.

[6] Adya, A., Wattenhofer, R. P., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., and Theimer, M., "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of the* 5th *Symposium on Operating Systems Design and Implementation*, 2002.

[7] Agrawal, G. and Jalote, P., "Coding based replication schemes for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 3, pp. 240–251, 1995.

[8] Agrawal, G., "Availability of coding based replication schemes," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 1992.

[9] Alvisi, L., Malkhi, D., Pierce, E., Reiter, M., and Wright, R. N., "Dynamic Byzantine quorum systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2000.

[10] Alvisi, L., Malkhi, D., Pierce, E., and Reiter, M. K., "Fault detection for Byzantine quorum systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 9, pp. 996–1007, 2001.

[11] Anderson, R. J., "The Eternity service," in *Proceedings of the* 1st *International Conference on Theory and Application of Cryptography (Pragocrypt)*, 1996.

[12] Backes, M., Cachin, C., and Strobl, R., "Proactive secure message transmission in asynchronous networks," in *Proceedings of the* 22nd *ACM Symposium on Principles of Distributed Computing*, 2003.

[13] Barak, B., Halevi, S., Herzberg, A., and Naor, D., "Clock synchronization with faults and recoveries," in *Proceedings of the* 19th *ACM Symposium on Principles of Distributed Computing*, 2000.

[14] Benaloh, J. and Leichter, J., "Generalized secret sharing and monotone functions," in *Crypto*, 1988.

[15] Blakley, G. R., "Safeguarding cryptographic keys," in *Proceedings of the National Computer Conference*, 1979.

[16] Cachin, C., Kursawe, K., Lysyanskaya, A., and Strobl, R., "Asynchronous verifiable secret sharing and proactive cryptosystems," in *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, 2002.

[17] Canetti, R., Gennaro, R., Herzberg, A., and Naor, D., "Proactive security: Long term protection against breakins," *RSA Laboratories' Cryptobytes*, vol. 3, no. 1, 1997.

[18] Canetti, R., Halevi, S., and Herzberg, A., "Maintaining authenticated communication in the presence of break-ins," in *Proceedings of the* 16th *ACM Symposium on Principles of Distributed Computing*, 1997.

[19] Canetti, R. and Herzberg, A., "Maintaining security in the presence of transient faults," in *Crypto*, 1998.

[20] Castro, M. and Liskov, B., "Practical Byzantine fault tolerance," in *Proceedings of the* 3rd *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[21] Castro, M. and Liskov, B., "Proactive recovery in a Byzantine fault tolerant system," in *Proceedings of the* 4th *Symposium on Operating Systems Design and Implementation*, 2000.

[22] Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., and Yianilos, P., "A prototype implementation of archival intermemory," in *Proceedings of the* 4th *ACM International Conference on Digital Libraries*, 1999.

[23] Chor, B., Goldwasser, S., Micali, S., and Awerbuch, B., "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *Proceedings of the* 26th *IEEE Symposium on Foundations of Computer Science*, 1985.

[24] Chow, C. S. and Herzberg, A., "Network randomization protocol: A proactive pseudorandom generator," in *Proceedings of the* 5th *USENIX Unix Security Symposium*, 1995.

[25] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W., "Freenet: A distributed anonymous information storage and retrieval system," in *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[26] Deswarte, Y., Blain, L., and Fabre, J. C., "Intrusion tolerance in distributed computing systems," in *Proceedings of the* 14th *IEEE Symposium on Security and Privacy*, 1991.

[27] Dingledine, R., Freedman, M. J., and Molnar, D., "The Free Haven Project: Distributed anonymous storage service," in *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[28] Feldman, P., "A practical scheme for non-interactive verifiable secret sharing," in *Proceedings of the* 28th *IEEE Symposium on Foundations of Computer Science*, 1987.

[29] Fischer, M. J., Lynch, N. A., and Paterson, M. S., "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[30] Frankel, Y., Gemmel, P., Mackenzie, P. D., and Yung, M., "Proactive RSA," in *Crypto*, 1997.

[31] Garg, V. K. in *Elements of Distributed Computing*, ch. 26.3, John Wiley and Sons, 2002.

[32] Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K., "Efficient Byzantine-tolerant erasure-coded storage," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[33] Herlihy, M. and Tygar, J. D., "How to make replicated data secure," in *Crypto*, 1987.

[34] Herzberg, A., Jakobsson, M., Jarecki, S., Krawczyk, H., and Yung, M., "Proactive public key and signature systems," in *Proceedings of the* 4th *ACM Symposium on Computer and Communications Security*, 1997.

[35] Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M., "Proactive secret sharing or: How to cope with perpetual leakage," in *Crypto*, 1995.

[36] Ito, M., Saito, A., and Nishizeki, T., "Secret sharing scheme realizing general access structure," in *Proceedings of the IEEE Global Communication Conference*, 1987.

[37] Iyengar, A., Cahn, R., Jutla, C., and Garay, J., "Design and implementation of a secure distributed data repository," in *Proceedings of the* 14th *IFIP International Information Security Conference*, 1998.

[38] Kong, L., Manohar, D. J., Subbiah, A., Sun, M., Ahamad, M., and Blough, D. M., "Agile Store: Experience with quorum-based data replication techniques for adaptive Byzantine fault tolerance," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 2005.

[39] Kong, L., Subbiah, A., Ahamad, M., and Blough, D., "A reconfigurable Byzantine quorum approach for the Agile Store," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 2003.

[40] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B., "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the* 9th *ASPLOS*, 2000.

[41] Kulesza, K. and Kotulski, Z., "On secret sharing schemes with extended capabilities," in *Proceedings of the Regional Conference on Military Communication and Information Systems*, 2002.

[42] Lakshmanan, S., Ahamad, M., and Venkateswaran, H., "Responsive security for stored data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 9, pp. 818–828, 2003.

[43] Lamport, L., "On interprocess communication, part 1: Basic formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[44] Malkhi, D. and Reiter, M., "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[45] Malkhi, D. and Reiter, M., "Secure and scalable replication in Phalanx," in *Proceedings of the* 17th *IEEE Symposium on Reliable Distributed Systems*, 1998.

[46] Malkhi, D., Reiter, M., Tulone, D., and Ziskind, E., "Persistent objects in the Fleet system," in *Proceedings of the* 2nd *DARPA Information survivability conference and exposition (DISCEX)*, 2001.

[47] Marsh, M. A. and Schneider, F. B., "CODEX: A robust and secure secret distribution system," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 34–47, 2004.

[48] Martin, J. P., Alvisi, L., and Dahlin, M., "Small Byzantine quorum systems," in *Proceedings of the International Conference on Dependable Systems & Networks*, 2002.

[49] Matthews, T., "Suggestions for random number generation in software," *RSA Laboratories' Bulletin*, January 1996.

[50] Naor, M. and Wool, A., "Access control and signatures via quorum secret sharing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 9, pp. 909–922, 1998.

[51] Ostrovsky, R. and Yung, M., "How to withstand mobile virus attacks," in *Proceedings of the* 10th *Symposium on the Principles of Distributed Computing*, 1991.

[52] Patterson, D. A., Gibson, G. A., and Katz, R. H., "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[53] Pedersen, T. P., "Non-interactive and information-theoretic secure verifiable secret sharing," in *Crypto*, 1991.

[54] Rabin, M., "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM*, vol. 38, no. 2, pp. 335–348, 1989.

[55] Rabin, T., "A simplified approach to threshold and proactive RSA," in *Crypto*, 1998.

[56] Reischuk, R., "A new solution to the Byzantine generals problem," *Information and Control*, vol. 64, no. 1-3, pp. 23–42, 1985.

[57] Shamir, A., "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[58] Stinson, D. R. and Wei, R., "Unconditionally secure proactive secret sharing scheme with combinatorial structures," in *Proceedings of the* 6th *International Workshop on Selected areas in cryptography*, 1999.

[59] Strunk, J. D., Goodson, G. R., Scheinholtz, M. L., Soules, C. A. N., and Ganger, G. R., "Self-securing storage: Protecting data in compromised systems," in *Proceedings of the* 4th *Symposium on Operating Systems Design and Implementation*, 2000.

[60] Subbiah, A. and Blough, D. M., "An approach for fault tolerant and secure data storage in collaborative work environments," in *Proceedings of the Workshop on Storage Security and Survivability (StorageSS)*, 2005.

[61] Thambidurai, P. and Park, Y.-K., "Interactive consistency with multiple failure modes," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 1988.

[62] Waldman, M., Rubin, A. D., and Cranor, L. F., "Publius: A robust, tamper-evident, censorship-resistant web publishing system," in *Proceedings of the* 9th *Usenix Security Symposium*, 2000.

[63] Wiseman, Y., Schwan, K., and Widener, P., "Efficient end to end data exchange using configurable compression," in *Proceedings of the* 24th *International Conference on Distributed Computing Systems*, 2004.

[64] Wong, T. M., *Decentralized recovery for survivable storage systems*. PhD thesis, Carnegie Mellon University, 2004.

[65] Wong, T. M., Wang, C., and Wing, J. M., "Verifiable secret redistribution for archive systems," in *Proceedings of the* 1st *International IEEE Security in Storage Workshop*, 2002.

[66] Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliççöte, H., and Khosla, P. K., "Survivable information storage systems," *IEEE Computer*, vol. 33, no. 8, pp. 61–68, 2000.

[67] Zhou, L., *Towards Fault-Tolerant and Secure On-Line Services*. PhD thesis, Cornell University, 2001.

[68] Zhou, L., Schneider, F. B., and Ranesse, R., "COCA: A secure distributed online certification authority," *ACM Transactions on computer systems*, vol. 20, no. 4, 2002.

[69] Zhou, L., Schneider, F. B., and Ranesse, R., "APSS: Proactive secret sharing in asynchronous systems," *ACM Transactions on Information and System Security*, vol. 8, no. 3, 2005.