Using Byzantine Quorum Systems to Manage Confidential Data*

Arun Subbiah†, Mustaque Ahamad‡, Douglas M. Blough†

† School of Electrical and Computer Engineering
‡ College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA
{arun,dblough}@ece.gatech.edu, mustaq@cc.gatech.edu

Technical Report: GIT-CERCS-04-13
March 31, 2004

Abstract

This paper addresses the problem of using proactive cryptosystems for generic data storage and retrieval. Proactive cryptosystems provide high security and confidentiality guarantees for stored data, and are capable of withstanding attacks that may compromise all the servers in the system over time. However, proactive cryptosystems are unsuitable for generic data storage uses for two reasons. First, proactive cryptosystems are usually used to store keys, which are rarely updated. On the other hand, generic data could be actively written and read. The system must therefore be highly available for both write and read operations. Second, existing share renewal protocols (the critical element to achieve proactive security) are expensive in terms of computation and communication overheads, and are time consuming operations. Since generic data will be voluminous, the share renewal process will consume substantial system resources and cause a significant amount of system downtime. Two schemes are proposed that combine Byzantine quorum systems and proactive secret sharing techniques to provide high availability and security guarantees for stored data, while reducing the overhead incurred during the share renewal process. Several performance metrics that can be used to evaluate proactively-secure generic data storage schemes are identified. The proposed schemes are thus shown to render proactive systems suitable for confidential generic data storage.

Keywords: Distributed data storage, Byzantine quorum systems, proactive secret sharing, replication, confidentiality

1 Introduction

The storage of sensitive information has been studied extensively in various contexts ranging from cryptographic keys [1] to generic data [2]. Computing power, network bandwidth, and secondary storage capacities have meanwhile increased dramatically, and seem to show no signs of abatement. While this trend has certainly helped in providing more secure data storage services and higher capacities, it has also empowered attackers in compromising storage servers and gaining access to critical data. Also due to this trend, the scope of "sensitive" information has broadened from obviously sensitive information 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.

To a data storage service provider, the above arguments translate to providing high levels of security guarantees for generic data. Also of concern to the data storage service provider is the investment that needs to be made in terms of hardware and software in order to provide highly secure data storage services.

This paper addresses these issues, and analyzes the benefits offered by the combination of quorum systems and proactive secret sharing schemes - two techniques that are probably the best today in terms of providing

*This research was supported by the National Science Foundation under Grant CCR-0208655.

some of the security guarantees. Quorum systems are a replication-based technique, and offers high availability. Secret sharing schemes are coding techniques that provide a high level of confidentiality. Proactive secret sharing schemes have the ability to change the coded data at the different data servers in a distributed fashion, which must be done periodically, e. g. once a week. This forces an adversary to compromise a certain threshold number of servers within this time frame (a week) in order to learn the sensitive data or to make the system behave incorrectly, which greatly enhances the security of such schemes.

We do not use the classic solution of encrypting data using cryptographic keys and using proactive secret sharing schemes to keep the keys secure, because the generic data we are trying to protect could very well be cryptographic keys. Also, with encryption-based techniques, long term confidentiality is not guaranteed, as given enough time, an adversary can compromise a server and find the key given the plaintexts and ciphertexts. The adversary may also find two ciphertexts identical and conclude that their plaintexts must match. A workaround would be to have the user change his key when such threats arise and then re-encrypt all the data that was stored using the old key; however this is cumbersome. Providing confidentiality based on encryption calls for a key management infrastructure and the active participation of users, since they must own and protect their keys. The drawback of encryption-based techniques is that data confidentiality falls completely within the users' purview, and requires a key management policy when data is shared among multiple users.

Our contributions are as follows: We believe this is the first work that suggests the use of proactive cryptosystems that provide unconditional security for generic data storage (large amounts of data that can be actively read and written). In such a model, the efficiency of the protocols is of concern. We propose to integrate quorum systems with proactive secret sharing schemes to reduce the overheads induced by client protocols and the share renewal protocols to maintain proactive security. The share renewal protocols, in particular, are known to incur high computation and communication overheads, rendering them unsuitable for voluminous data storage uses. We present share renewal protocols that reduce these overheads and make them practical. We identify several performance metrics that can be used to characterize the efficiency of such systems. We propose two techniques that blend quorum systems with proactive cryptosystems, and describe the client protocols and the share renewal protocol, which is used to maintain proactive security, for both the techniques. We demonstrate that such solutions are a significant step forward in realizing practical generic-data storage systems that provide unconditional proactive security.

2 Related Work

Several works [3, 4, 5, 6, 7, 8, 9, 10, 11, 12] have emerged recently that consider the problem of providing secure distributed data storage services. The confidentiality of the stored data is provided either by encrypting the data with a key and storing the key also at the store using secret sharing [1, 13], or secret sharing the data itself, or a combination of both.

Storage algorithms that use cryptographic keys do not provide long-term confidentiality, as mentioned in the Introduction. Many algorithms will be broken with the development of higher computing power. A testimony to this is the continued increase in the required key sizes for such algorithms. Increasing the required key size is equivalent to changing the key, which would require a user or a trusted third party to decrypt and re-encrypt all confidential data with the new key.

Secret sharing schemes are another approach to confidentiality. Most works that consider storing confidential data use imperfect secret sharing schemes, such as error correcting codes and Rabin's IDA [14] algorithm, where the knowledge of fewer than the threshold number of shares can reveal some information of the encoded data. Such coding algorithms allow savings in storage space, but do not, by themselves, provide long-term confidentiality. An adversary has the entire lifetime of the system to attack the servers and learn a threshold number of shares. To overcome this, the schemes 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 share renewal scheme for imperfect secret sharing has been developed.

Recent works [15, 16] have meanwhile optimized replication based techniques when the fault threshold can be varied, while imperfect secret sharing schemes assume a fixed fault threshold. In general, there is a tradeoff between performance and security. The ability to vary the fault threshold depending on the current threat levels will greatly enhance the performance of the distributed storage service. Changing the fault threshold in imperfect secret sharing schemes would require a special, secure entity (not the user) to decode all the data and re-encode it, which is not practical. On the other hand, perfect secret sharing schemes, such as Shamir's [1] scheme, allow the fault threshold to be varied [17].

Several works have integrated quorum systems [18] and secret sharing in various ways [19, 20, 2, 21]. In [19, 20], quorum systems are used with Rabin's IDA scheme to improve the system availability. [2] presents a scheme where data is encrypted using a key, and both are managed using quorum systems. The data is stored in replicated form in a quorum, while the key is stored using secret sharing. Thus, availability of the data is increased, but no verifiable secret sharing or share renewal is done to provide long-term confidentiality. Also, the inherent drawbacks due to encryption are present. [21] considers a quorum system where the shares of a secret are stored at all the servers, and a quorum of shares are needed to recover the secret. The secrets are read-only in nature, and quorum properties are used to grant or revoke access rights and do not directly affect the security of the secret. Share renewal and proactive security are not discussed because the main motivation is to use quorums for access control while keeping the permissions secret.

Other works [22] have considered the use of coteries and access structures with secret sharing. Here, the problem is that of having distinct sets of servers able to collectively reconstruct the data, while some sets of servers should not be able to. Though the valid access structures could be called quorums, the analysis is in terms of reducing the number of shares per server and does not consider the quality of data storage service.

Some works such as [23] that do consider using perfect secret sharing schemes consider only archival storage, where data is never re-written after it is created. Most works consider awarding confidentiality guarantees only to archival storage. [24] considers perfect secret sharing schemes for read/write data using dissemination of share replicas, but provides only weak consistency. It is also not clear if and how share renewal can be integrated with their approach.

Long-term confidentiality of data is provided by proactive secret sharing schemes, which were first considered by Ostrovsky and Yung in [25]. In that paper, the mobile adversary model was introduced. The mobile adversary is assumed to be able to compromise all the servers given enough time, but can compromise no more than a threshold number of servers in a time period. Herzberg, et al., [26] gave a robust proactive secret sharing scheme. Proactive secret sharing schemes use a combination of verifiable secret sharing [27] and verifiable share renewal. Proactive secret sharing schemes for synchronous networks can be found in [28] and the references therein.

3 System Model

We assume a set of n data servers that together provide the data storage service. A separate set of entities, called clients, read and write to a quorum or group of servers to manage their data.

3.1 Fault Model

We assume that the data servers can experience Byzantine, or arbitrary, failures. This means that faulty servers can act in an arbitrary fashion, which includes behavior of compromised servers. Faulty clients are taken care of using access control techniques and verifiable secret sharing schemes. Access control prevent faulty or malicious clients from using the specified read and write protocols to perform unauthorized data access. Verifiable secret sharing safeguards against malicious clients that may write inconsistent shares for the same data object, or not write to a complete write quorum (partial writes).

3.2 Communication Model

Each server has two network interfaces. One connects it to an authenticated, unreliable and asynchronous network, denoted by RA, and the other network interface connects it to an authenticated, synchronous broadcast channel, denoted by BC. Clients connect to servers using the network RA. If two fault-free parties S and R are connected by such a network RA, then if S repeatedly sends a packet to R, the packet will eventually be received by R. R will be able to verify the integrity and authenticity of the received packet. No bound is placed on message transmission times.

The authenticated, synchronous broadcast channel BC links all the servers together, and no other entity is legally allowed to connect to this channel. Channel BC is used exclusively for share renewal and in maintaining proactive security. It has the property that messages sent on it instantly reach every party connected to it. The servers can thus be synchronized to a common clock. The zero-delay broadcast channel assumption is heavily used in schemes that provide proactive security, such as Herzberg's [26]. Recently, proactive security in asynchronous systems has been considered in [29, 30]. Adapting our analyses and solutions to fit these models is an area for future work.

3.3 Time and Clock Model

Using the common broadcast channel BC, all the servers are synchronized to a common global clock. Time is divided into *time periods*, as determined by the global clock. Servers participate in a share renewal protocol at the beginning of each time period.

3.4 Adversary Model

We assume the *mobile* adversary model [25]. In this model, an adversary is capable of compromising all the servers in the system given enough time. Compromised servers are assumed to be corrected in a timely fashion so that there are no more than a threshold number of compromised servers in a time period. We assume that correction of compromised servers can be done by a simple reboot operation or something similar, which will return the server to a state that it would have reached had it not been compromised by the adversary. To compromise a corrected server, an adversary must launch his attack all over again. In other words, the server correction process completely removes any security trapdoors that the adversary might have installed while the server was compromised. Herzberg's PSS scheme [26] provides explicit mechanisms by which a majority of correct servers can detect and alert the system management of any misbehaving server during the share renewal phase. In [15], a fault detection algorithm based on quorum protocols is presented. Other methods of detecting compromised servers include intrusion detection techniques.

Besides timely correction of compromised servers, share renewal must be done periodically so that an adversary will be unable to eventually recover a threshold number of consistent shares. Share renewal is done at the time period boundaries. Servers compromised during the share renewal phase are assumed to have been compromised in the time periods just before and after the share renewal phase.

Correct (fault-free) servers must reliably erase data when required. Otherwise, when correct servers get compromised, an adversary may acquire "old" data, which for one defeats the purpose of share renewal.

Cryptographic techniques allow the detection of messages corrupted by adversaries as well as spoofing of messages by an adversary. It is assumed that an adversary cannot prevent uncompromised servers from receiving messages destined to them. Also, an adversary cannot employ denial-of-service attacks. The adversary is assumed to be aware of all publicly-known data such as the algorithm used for secret sharing and the quorum sizes.

The adversary is assumed to be computationally bounded, meaning he cannot break the underlying cryptographic algorithms used for public key encryption and verifiable secret sharing.

3.5 Cryptography Assumptions

It is assumed that any two parties (client-server and server-server) establish a temporary symmetric-key to encrypt and decrypt messages during a session. These keys must be changed during the time period boundaries and when compromised servers are corrected. Details of these schemes is beyond the scope of this paper. The solutions to these problems can be obtained from [26].

4 Preliminaries

In this section, we review Byzantine quorum systems, the secret-sharing scheme, and the share renewal scheme. Readers who are aware of these concepts can skip to the next section without loss of continuity.

4.1 Byzantine Quorum Systems

Quorum systems are replication-based techniques. 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 [31], the minimum overlap between any two quorums is increased sufficiently in order to accomodate Byzantine failures.

A brief description of the read and write protocols for generic data is as follows. To read data, the client picks the value that has the highest timestamp and seconded by at least $b+1$ servers from a read quorum of responses. To write data, the client executes a read on the data timestamp, increments it to a higher value, and then writes the data value and the new timestamp to a write quorum of servers. The read and write quorums must therefore overlap in at least $2b+1$ servers (for generic data). Such protocols provide safe variable semantics [32].

In [31], it has been shown that the minimum number of servers required to tolerate b Byzantine server failures for generic data is $4b+1$. Threshold masking quorums for generic data use a write quorum size of $\lceil \frac{n+2b+1}{2} \rceil$ and a read quorum size of $\lfloor \frac{n+2b+1}{2} \rfloor$ servers. For self-verifying data, the minimum number of servers required is $3b+1$, with the choice of write quorum size being $\lceil \frac{n+b+1}{2} \rceil$ and the read quorum size being $\lfloor \frac{n+b+1}{2} \rfloor$ servers.

4.2 Shamir's Secret-Sharing Scheme

The secret-sharing algorithm used in this paper is due to Shamir [1]. A brief outline of the algorithm is as follows:

Consider the problem of generating n shares of a secret S such that any k shares can yield S. Such a scheme is called a (k,n) threshold scheme. Shamir's scheme has the additional property that any $k-1$ shares do not give any information on S.

The secret S and all operations described are performed in Z_{p}, where p is prime and greater than n. Construct a polynomial $f(x) = a_{\mathrm{k}-1}x^{\mathrm{k}-1} + a_{\mathrm{k}-2}x^{\mathrm{k}-2} + ... + a_1 x + S$ where the a_{i}'s are arbitrary coefficients and S is the secret to be shared. The i^{th} share, s_{i}, is given by $f(i)$.

To retrieve the secret S from any k shares, Lagrange interpolation is used. Denote the set of shares (i, s_{i}) to be used for reconstructing the secret S by A. The secret S is given by

$$S = \sum_{\mathrm{i}\epsilon\mathrm{A}} m_{\mathrm{i}} s_{\mathrm{i}} \quad \text{where} \quad m_{\mathrm{i}} = \prod_{\mathrm{j}\epsilon\mathrm{A},\mathrm{j}\neq\mathrm{i}} \frac{j}{j-i} \tag{1}$$

4.3 Verifiability of shares

Feldman's scheme is used to add verifiability in many share renewal works such as [26]. The security is due to the difficulty in computing discrete logarithms over a finite field by a computationally-bounded adversary.

Let q be a prime number such that $q = mp + 1$, where m is a small integer and p is the prime number used in Shamir's scheme. Let g be an element of Z_q of order p. To demonstrate Feldman's scheme as applied to Shamir's scheme, consider a client responsible for generating shares of a secret. When a client constructs the polynomial $f(x)$ to distribute the secret S, he computes *witnesses* g^{a_1}, g^{a_2}, ..., $g^{\mathrm{a}_{\mathrm{m}-1}}$, and g^{S}, all of which are computed in the finite field Z_q, and makes these values public. The value g is also made public. Server i verifies that its share s_i of secret S is valid by verifying that

$$g^{s_\mathrm{i}} \equiv (g^{\mathrm{S}})(g^{\mathrm{a}_1})^{\mathrm{i}}(g^{\mathrm{a}_2})^{\mathrm{i}^2}...(g^{\mathrm{a}_{\mathrm{k}-1}})^{i^{k-1}}(\mathrm{mod}\ q)$$

If the above relation holds, then server i broadcasts on the synchronous common broadcast channel BC to all other servers that it found the share it received to be valid. If all severs find the shares they received to be valid with the same set of witnesses, then they commit their shares and the process is said to be complete.

When a client retrieves the secret, the witnesses with the highest timestamp and returned by at least $b+1$ servers are the correct ones, and individual shares returned by servers can be checked with these witnesses to distinguish correct and incorrect shares.

Some information on x given g and g^{x} can be obtained. For a discussion on how to overcome this, the reader is directed to [26].

4.4 Herzberg's Proactive Secret Sharing Scheme

Herzberg's Proactive Secret Sharing Scheme (PSS) [26] is used for periodic share renewal. The synchronous common broadcast channel BC is used for this purpose.

PSS is divided into three stages. The first stage is that of renewing the keys used by the servers. If this were not done, then even after a compromised server is corrected, an adversary can decode messages addressed to the compromised server, and also inject messages on behalf of the now-corrected server.

Following the key renewal stage, servers engage in a share recovery protocol. This allows corrected servers that had their shares deleted by an adversary to recover them. The share recovery protocol can successfully regenerate shares provided at least $b+1$ servers which have the latest shares participate.

The final stage is that of share renewal. The shares of each secret or data object are renewed by having each server generate shares of "0" using a (k, n) Shamir's scheme for every other server and then distributing it verifiably and securely. Upon receiving a share of "0" and after the verifiability checks have passed, servers add it to their current shares of the secret, which were also generated using a (k, n) Shamir's scheme. It is easy to see that the encoded secret will still remain the same due to the linearity in polynomial addition.

5 Performance Metrics

We describe below the performance metrics we use to evaluate and compare distributed storage algorithms.

1. **Availability:** Traditionally, availability is measured in the presence of crash failures. Therefore, when calculating this metric, we allow crash faults in addition to Byzantine faults. The availability is defined as the probability of finding a live read or write quorum given that each server can fail independently with a certain probability. The assumption of independent failures may not always hold, as attacks such as that of Denial-of-Service can cause highly correlated unavailability of servers. As noted in [33], the exact availability values may therefore not be important, but relative values are. A relatively higher availability indicates tolerance to a relatively larger number of server failures.

2. **Minimum system size needed to tolerate a given number of faulty servers**

3. **Size of read and write quorums**

4. **Storage blowup factor:** is the ratio of the average storage space taken at each server per data object to the size of the data object.

5. **Load:** is defined as the probability that the busiest server is accessed during a request.

6. **Confidentiality:** is defined as the probability that not more than the threshold number of consistent shares can be obtained by an adversary.

7. **Computation overhead:** Servers incur computational overhead during writes and share renewal, while clients incur computational overhead during reads and writes. The computation overhead is measured in terms of the number of finite-field multiplication and exponentiation operations. Other operations such as additions, encryption, and hash computations are assumed to incur relatively negligible overhead.

8. **Communication and message complexities:** Communication complexity is the total number of bits exchanged during a protocol, while message complexity is the number of messages exchanged. For clients, these metrics are measured at the client network interface. For writes, only the overhead during the actual write is considered; timestamp reads are not taken into account. For servers, the overheads are measured as the total number of bits or messages sent on the channel BC.

6 Proposed Solutions

We describe our two proposed techniques in this section. To put things in perspective, consider the benefits and drawbacks of a pure quorum systems approach and a pure secret sharing approach for a system size of 44 servers and a fault threshold b of 3. If the probability of a server being compromised or crashed is 0.1, then the availability of read and write quorums in the pure quorum systems approach is at least 7 nines ($1 - 10^{-7}$), while the availability of a write quorum in a pure secret sharing approach is a very small 0.347, because shares must be written to *almost all* the servers in the system. On the other hand, the confidentiality offered by pure quorum systems is an unacceptably small $9.7\text{x}10^{-3}$, while the pure secret sharing approach offers a confidentiality of 3 nines (0.999), assuming the probability that a server is compromised in a *time period* is 0.01. Achieving a tradeoff between these two extremes is one of the principal motivations behind our proposed solutions, described below.

We do not describe the complete Herzberg's PSS scheme because the key renewal phase of PSS is independent of the actual data storage algorithm used, and the lost share recovery protocol of PSS is required only to recover lost shares. The share renewal part of Herzberg's scheme is the most intensive part of the protocol, and due to space constraints, we focus only on this aspect of PSS. Also, the verifiable sharing and share renewal protocols rely on all servers in the quorum being honest. When this is not the case, servers broadcast accusations against other servers and decide distributively if a server is faulty. Details of this method can be found in [26]. For brevity, we do not describe how these techniques coexist with our protocols.

6.1 Method RegularSharing

6.1.1 Description

In this method, a $(b+1)$-threshold Shamir's scheme is used to give each server in a write quorum a distinct share, unlike in "pure" quorum systems where data is replicated across a quorum of servers. For a fault threshold of b, at least $n = 4b+1$ servers are required. Quorums are picked randomly (regular quorums). The read and write quorum sizes must be at least $3b+1$ and not more than $n-b$ servers, and the read and write quorums must overlap in at least $2b+1$ servers. Thus, even in the presence of b Byzantine failures, there will be at least $b+1$ servers that will return values written in the last write. Feldman's VSS scheme is used for verifiability, and Herzberg's PSS is used for proactive security. A read quorum size of $q_\text{r} = \lfloor \frac{n+2b+1}{2} \rfloor$ and a write quorum size of $q_\text{w} = \lceil \frac{n+2b+1}{2} \rceil$ servers is assumed.

6.1.2 Read and Write Protocols

To write a data object V, the client uses a $(b+1)$-threshold Shamir's scheme to generate q_{w} shares of V. Let the polynomial chosen for this purpose be $f(x) = a_{\mathrm{b}}x^{\mathrm{b}} + a_{\mathrm{b}-1}x^{\mathrm{b}-1} + ... + a_1x + v \pmod p$. A server in the write quorum with id i will receive the share given by $f(i)$. From Feldman's scheme, the witnesses are $g^{a_{\mathrm{b}}}, g^{a_{\mathrm{b}-1}}, ..., g^{a_1}$, and g^{v}. The client sends each server in the write quorum its share and all the witnesses securely. The complete write and read protocols followed by the client is shown in Figure 1. The share stored at server i for data object V is denoted by $s_{\mathrm{V_i}}$. The timestamp of data object V is denoted by TS_{V}.

Upon receiving a write request, each server i in the write quorum checks if

$$g^{\mathrm{s_i}} = g^{\mathrm{v}}.(g^{\mathrm{a_1}})^{\mathrm{i}}.(g^{\mathrm{a_2}})^{\mathrm{i}^2}...(g^{\mathrm{a_b}})^{\mathrm{i}^{\mathrm{b}}} \pmod q \tag{2}$$

and if the new timestamp specified by the client for V is higher than the timestamp it has stored locally.

If these two checks are passed, server i is said to find the write request *valid*, and it broadcasts an acceptance message containing the timestamp and the hash of all the witnesses. The algorithm used to compute the hash is assumed to be collision free. If a *write quorum* of servers find the request with the same set of witnesses and timestamp valid, then servers replace their current shares, witnesses, and timestamp with the new values. Since the broadcast channel BC is used, servers can easily identify which servers belong to the latest write quorum and which do not. Servers that belong to the latest write quorum record the identities of other servers that are part of the write quorum, while servers that are not part of the write quorum delete their shares and the associated set of witnesses. The message and communication complexities on the channel BC during writes are therefore q_{w} messages and $160q_{\mathrm{w}}$ bits, where it is assumed that the length of a hash is 160 bits.

To read an object, a client obtains the timestamp, shares, and witnesses from a read quorum of servers. The client first determines the correct set of witnesses from the set of responses returned by at least $b+1$ servers that have the highest timestamp and with the same set of witnesses. Then, the shares returned with this highest timestamp are checked against the corresponding witnesses (equation 2), and if the check is passed, the received share is valid. From $b+1$ valid shares, the secret can be reconstructed because a $(b+1)$-threshold Shamir's scheme was used. The complete read protocol followed by the client is also given in Figure 1.

The read and write protocols implement safe variable semantics [32], and satisfy the following theorem:

Theorem 1 *For a given data object, a read operation that is not concurrent with any write operation returns the value written in the last write operation, in some serialization of all earlier write operations.*

Proof: A non-faulty server will accept a write only if the new timestamp of the object is higher than the timestamp it has stored locally, and if all servers in a write quorum agree on the timestamp and the consistency of the new shares. If the write is accepted, the correct server overwrites the current value of the data object's share, timestamp, and witnesses with the new values specified in the write. Feldman's VSS scheme, along with the synchronous broadcast channel assumption, ensures that a write quorum (barring faulty servers) will commit the write with the same timestamp and witnesses.

When a client executes a read that is not concurrent with a write, the read quorum will intersect the write quorum used in the last successful write operation in at least $2b+1$ servers. Since a maximum of only b servers may be faulty, the set of responses containing the highest timestamp returned by at least $b+1$ servers has a response at least from one correct server, which must have been a member of the last write quorum. This set of responses with the highest timestamp and returned by at least $b+1$ servers will contain the same set of witnesses and consistent shares for these witnesses. Since at least $b+1$ latest shares will thus be obtained, the client can easily recover the data object because a $b+1$ threshold scheme was used. ∎

6.1.3 Share Renewal

For share renewal, Herzberg's scheme [26] is used. For a given object, the protocol is run only at the write quorum of servers that have the shares of that object. Recall that servers delete old shares during the VSS

Read Protocol
1. Retrieve a read quorum of responses of the form $< TS_{\mathrm{V}}$, server id $i, s_{\mathrm{Vi}}, < witnesses >>$.
    Overhead: q_{r} *messages,* $q_{\mathrm{r}}(b+2)l$ *bits*
2. Choose the responses returned by at least $b+1$ servers that have the highest data timestamp
    If no such data timestamp exists, return *null.*
3. The set of witnesses contained in the selected set of responses and returned by at least $b+1$ servers is the correct set of witnesses. If no such correct set of witnesses can be found, return *null.*
4. Choose $b+1$ shares from the selected responses that are consistent with the witnesses found in step (3) above (equation 2). *Overhead: (b+1).(b+1) exps, b(b+1) mults*
    If no such set of $b+1$ shares can be found, return *null.*
5. Construct V from the selected $b+1$ shares. *Overhead: 0 exps, b+1 mults*

Write Protocol
1. Retrieve a read quorum of responses of the form $< TS_{\mathrm{V}} >$.
2. Choose the highest timestamp returned by at least $b+1$ servers.
3. Choose a timestamp ($TS_{\mathrm{V}}(new)$) higher than the timestamp determined above and not previously chosen by this client.
4. Pick a write quorum WQ, and, using a $b+1$-threshold Shamir's scheme, generate a share for each server in WQ. *Overhead: 0 exps,* bq_{w} *mults*
5. Compute the new set of witnesses for data object V. *Overhead: b+1 exps, 0 mults*
6. To each server i in the chosen write quorum WQ, write $< TS_{\mathrm{V}}(new), s_{\mathrm{Vi}}, < witnesses >>$.
    Overhead: q_{w} *messages,* $(b+2)lq_{\mathrm{w}}$ *bits*

Figure 1: Read and Write Protocols for Method RegularSharing

protocol followed during client writes, and are aware of the write quorums used. Herzberg's share renewal protocol adapted to Method RegularSharing is shown in Figure 2. $\mathrm{ENC}_{\mathrm{ij}}(M)$ denotes the encrypted version of the message M sent by server S_{i} to server S_{j} encrypted using a shared symmetric key. All other messages are sent unencrypted and are susceptible to eavesdropping. Faulty servers are not taken into account to simplify the presentation. The protocol only requires that a mechanism exists for a server to detect another server to be faulty. Once a suspicion arises, it must be solved using a distributed protocol such as the one given in [26].

6.1.4 Evaluation Of Perfomance Metrics

1. **Availability:** If the probability with which a server is either Byzantine-faulty or crashed is p, and is independent of other server failures, then the availability of a read quorum is given by $\sum_{\mathrm{f}=0}^{\mathrm{n}-\mathrm{q_r}} \begin{pmatrix} n \\ f \end{pmatrix} p^{\mathrm{f}}(1-p)^{\mathrm{n}-\mathrm{f}}$, and the availability of a write quorum is given by $\sum_{\mathrm{f}=0}^{\mathrm{n}-\mathrm{q_w}} \begin{pmatrix} n \\ f \end{pmatrix} p^{\mathrm{f}}(1-p)^{\mathrm{n}-\mathrm{f}}$, where $q_{\mathrm{w}} = \lceil \frac{n+2b+1}{2} \rceil$ and $q_{\mathrm{r}} = \lfloor \frac{n+2b+1}{2} \rfloor$.

2. **Minimum number of servers:** At least $4b+1$ servers are required to tolerate b Byzantine server faults [31].

3. **Size of read and write quorums:** for a given fault threshold b are $q_{\mathrm{r}} = \lfloor \frac{n+2b+1}{2} \rfloor$ and $q_{\mathrm{w}} = \lceil \frac{n+2b+1}{2} \rceil$ respectively.

4. **Storage blowup factor:** The size of a share is the same as that of the secret. The witnesses used due to Feldman's scheme take up an additional storage space equal to $(b+1)$ times the size of an object. This corresponds to a storage blowup factor of $(b+2)$ per data object. For all data objects put together, the

Each server S_i that contains a share of data object V executes:

1. Pick b random numbers $(\delta_1 ... \delta_b)$ from Z_p to define the polynomial $\delta_i(x) = \delta_{i1}x + \delta_{i2}x^2 + ... + \delta_{ib}x^b$. Compute $\epsilon_{im} = g^{\delta_{im}} (\bmod\ q)$, $1 \leq m \leq b$. *Overhead: b exps, 0 mults*
2. Compute $u_{ij} = \delta_i(j) (\bmod\ p), j \in WQ_V$. *Overhead: 0 exps, bq_w mults*
3. Broadcast $(i, TS_V, \{\epsilon_{im}\}_{m=1,...,b}, \{\text{ENC}_{ij}(u_{ij})\}_{j \in WQ_V, j \neq i})$. *Overhead: 1 message, (q_w+b-1)l bits*
4. If at least $(q_w - b)$ servers in the system have a timestamp TS_V higher than what is locally stored, then delete local share of V and all witnesses associated with V, and exit the protocol.
5. From all messages broadcast by other servers that have the same timestamp as what is stored locally, verify the correctness of shares received using $g^{u_{ji}} = (\epsilon_{j1})^{i} (\epsilon_{j2})^{i^2} ... (\epsilon_{jb})^{i^b} (\bmod\ )q, j \neq i$
   Overhead: (b+1)(q_w-1) exps, (b-1)(q_w-1) mults
6. If the above check is passed by a write quorum of subshares, then broadcast an acceptance message.
   Overhead: 1 message, <negligible> bits
7. If a write quorum of servers sent acceptance messages then update local share by performing $s_{i,new} = s_{i,old} + \sum_{j \in WQ_V} u_{ji} (\bmod\ p)$, update witnesses $(witnesses)_{new} = (witnesses)_{old}$ x ϵ_j's, and erase all other variables used. *Overhead: 0 exps, bq_w mults*

Figure 2: The Share Renewal Protocol for Data Object V in Method RegularSharing

consumed storage space will only be $\frac{q_w}{n}$ of the total storage space because not all servers participate in the most recent write, and during the write old shares are deleted. The blowup factor is thus $(b+2)\frac{q_w}{n}$.

5. Load: If r denotes the percentage of requests that are reads, w denotes the percentage of requests that are writes $(r + w = 1)$, q_r denotes the size of the read quorum, q_w denotes the size of the write quorum, then the load on the system is given by $\frac{r.q_r + w.q_r + w.q_w}{(r+w+w)n}$. Simplifying, load $= \frac{q_r + w.q_w}{(1+w)n}$. The load caused by timestamp reads during writes has been taken into account in the above calculation.

6: Confidentiality: The compromise of $b + 1$ servers will cause the system to lose confidentiality. If the probability that a server is compromised is denoted by p, then the confidentiality of the system is the probability that not more than b servers are compromised, given by $\sum_{f=0}^{b} \left(\begin{array}{c} n \\ f \end{array} \right) p^f (1-p)^{n-f}$, assuming probabilities of server compromises are independent.

7. Computation overhead: During a write, the client performs $(b + 1)$ exponentiations in computing the witnesses for Feldman's VSS scheme, and b multiplications in computing each share, giving a total of bq_w multiplications for all servers in the write quorum. Each server in the write quorum performs Feldman's VSS, which involves $b + 1$ exponentiations and b multiplications.

During a read, servers do not incur any computation overheads. The client must check the correctness of shares against the latest witnesses using Equation 2, which involves $(b + 1)$ exponentiations and b multiplications per share. Since $b + 1$ shares are needed, the client must perform at least $(b + 1)^2$ exponentiations and $b(b+1)$ multiplications. In addition, to recover the secret, the client must multiply each share with a certain coefficient (Eqn 1) which incurs an overhead of $b + 1$ multiplications.

A server participating in a share renewal performs: in step (1) of the share renewal protocol, b exponentiations; in step (2), b multiplications per share, resulting in a overhead of bq_w multiplications; in step (5) of the protocol, $b + 1$ exponentations and $b - 1$ multiplications for all other servers in the write quorum, resulting in a total overhead of $(b + 1)(q_w - 1)$ exponentiations and $(b - 1)(q_w - 1)$ multiplications; and in step (7), b multiplications for every server in the write quorum, giving a total overhead of bq_w multiplications. Thus, there are $((q_w - 1)(b + 1) + b)$ exponentiations and $(q_w - 1)(b - 1) + 2q_w b$ multiplications involved during the share renewal process at each server in a write quorum for an object, where $q_w = \lceil \frac{n+2b+1}{2} \rceil$.

8. **Message and communication complexities:** During writes, a client sends a share and $(b+1)$ witnesses securely to each server in a write quorum. Thus, a total of $q_{\rm w}$ messages are sent. Since the length of the primes p and q (used for secret-sharing and verifiability respectively) are approximately the same, both are assumed to be l bits long. The communication complexity at the client is then $l(b+2)q_{\rm w}$ bits.

   As part of the VSS protocol during writes, each server in the write quorum broadcasts acceptance messages along with the hash of the witnesses on the broadcast channel BC. The total number of messages exchanged on the channel BC is $q_{\rm w}$ messages, and the total number of bits exchanged is $160q_{\rm w}$ bits, where it assumed that the length of a hash is 160 bits.

   During reads, a client receives a total of $q_{\rm r}$ responses or messages. Each message contains a share and the set of $(b+1)$ witnesses. The communication complexity is therefore $(b+2)lq_{\rm r}$ bits.

   During share renewal, each server that is part of the write quorum broadcasts two messages (step (3) and step (6) in Figure 2). Thus, the message complexity on the channel BC is $2q_{\rm w}$ messages. Since each server in a write quorum sends all other servers in the same write quorum a subshare and b witnesses, the total communication complexity on the channel BC is $(b+q_{\rm w}-1)lq_{\rm w}$ bits.

6.2 Method GridSharing

6.2.1 Description

This method is an effort to make the share renewal process simple. The servers are arranged in a grid consisting of $b+1$ rows, where b is the fault threshold. A $(b+1, b+1)$-Shamir's scheme is used, with each share being written to a write quorum of servers along a row, i.e., the $r^{\rm th}$ share is replicated across a write quorum of servers in row r. Shares are written to and read from quorums in each row.

Shares are made self-verifying using distributed fingerprints. For self-verifying data, at least $3b+1$ servers are required in each row [31], i.e., the minimum number of servers in the system is $n = (b+1)(3b+1)$. If there are c servers in each row, $c \geq 3b+1$, then for each row, the read quorum size is given by $\lfloor \frac{c+b+1}{2} \rfloor$ and the write quorum size is given by $\lceil \frac{c+b+1}{2} \rceil$.

When a data object V is created the first time, shares must be written to quorums in each row. Subsequent updates to V can be brought about by modifying only one share, thus reducing the size of the write quorum. Denote the share given to row r, $1 \leq r \leq b+1$, by $s_{\rm Vr}$. Then the value v stored in data object V is given by

$$v = m_1 s_{\rm V1} + m_2 s_{\rm V2} + ... + m_{\rm b} s_{\rm Vb} + m_{\rm b+1} s_{\rm V,b+1} \quad \text{where} \quad m_{\rm r} = \prod_{\rm j=\{1,...,(b+1)\},j\neq r} \frac{j}{j-r} \tag{3}$$

When a client wants to update v to v', only one of the shares in the above summation needs to be changed. For data object V, let the share distributed along the $(b+1)^{\rm th}$ row alone be changed from $s_{\rm V,b+1}$ to $s'_{\rm V,b+1}$. Thus,

$$v' = m_1 s_1 + m_2 s_2 + ... + m_{\rm b} s_{\rm b} + m_{\rm b+1} s'_{\rm V,b+1} \tag{4}$$

The modified share needs to be written only to a write quorum of servers in the $(b+1)^{\rm th}$ row.

Note that requiring the knowledge of V prior to making the update should not be viewed as an overhead. It is a well-known concept that overall efficiency is increased when only differences in versions are propagated instead of writing data in its entirety every time a change is made. This concept has been extended to secret sharing this way. The client always has the option of over-writing all the shares if it chooses to do so. We, however, analyze the performance of this method assuming updates are performed in the way described above. When the value of object V is changed from v to v', the client needs to be aware of only v and $s_{\rm V,b+1}$ in order to derive $s'_{\rm V,b+1}$.

The above method introduces the obvious threat that a user who has his access rights revoked may recover secret data later by the compromise of only one server. For example, a user may be aware of the values v and $s_{\rm V,b+1}$ for data object V. After his access rights get revoked, v may be changed to v' by a legitimate user.

The user who got his access rights revoked needs to compromise some server in row $b+1$ to find the modified share, and hence recover the new secret value v'. Periodic share renewal only partly solves this problem, as this vulnerability exists from the time access rights are revoked and some secret data is updated until the next share renewal phase. This vulnerability can be completely eliminated by having the next write after a revocation to take place at write quorums in all the rows. That is, the protocol to be followed is that of creating a data object the first time, except that the timestamp chosen will be the highest and not zero.

Since only $b+1$ shares are generated using a $b+1$ threshold scheme, there is no need to check for consistency between the shares and the secret. Hence, there is no need for Feldman's scheme in this method. The shares are made verifiable by storing the hash of each share in a hash vector denoted by $H_{\mathrm{V}}[\,]$. The hash vector itself must be managed using quorum systems for *generic* data. Hence, quorums along two rows of servers are used to manage the hash vector. For data object V, let the hash vector be stored in the b^{th} and $(b+1)^{\mathrm{th}}$ rows. The hash vector is managed by the servers themselves during writes / updates to V.

If the same row ($(b+1)^{\mathrm{th}}$) is contacted during updates to all data objects, then the load on this row will be heavy compared to other rows. To spread out the load evenly across all rows, a row is associated with each data object and is part of the data object's metadata information. Clients are assumed to know the row that needs to be contacted during updates to a given data object.

Herzberg's share renewal protocol is modified to use quorum systems for verifiability instead of Feldman's VSS to defend against active adversaries.

6.2.2 Read and Write Protocols

When a data object V is created the first time, the client initializes the timestamp, generates shares of V using a $(b+1, b+1)$-Shamir's scheme, and writes the timestamp and each share securely to a write quorum of servers in the respective rows.

To read the object V, a client reads from a read quorum of servers in each row the shares, and from read quorums in rows b and $b+1$, also the hash vector and the timestamp. The latest hash vector is the hash vector returned by at least $b+1$ servers that have the highest timestamp, and the correct share from each row is determined by comparing its hash against that stored in the hash vector.

During subsequent updates to V, if the client is not aware of b of the $b+1$ shares used to store V or the current value in V, then it executes a read on V. If the client does not know the current timestamp of V, it queries a read quorum of servers in rows b and $b+1$ for the timestamp. The current timestamp is the highest timestamp returned by at least $b+1$ servers. A timestamp higher than the current timestamp and not previously chosen by this client is chosen. In updates, only the $(b+1)^{\mathrm{th}}$ share (in case of data object V) is modified, and is updated at a write quorum of servers in row $b+1$, along with the new timestamp.

If the client decides to write all shares to update an object, then it first determines the current timestamp by querying read quorums in rows b and $b+1$. It then chooses a timestamp greater than the highest timestamp returned by at least $b+1$ servers, and writes the new timestamp and shares to a write quorum in the respective rows.

The complete write and read protocols followed by the client is shown in Figure 3. The share stored in row r for data object V is denoted by s_{Vr}. The timestamp of data object V is denoted by TS_{V}.

Upon receiving a write request to create a data object V, or when all the shares are modified during a write, servers broadcast the hash of the share they received along with the new timestamp on the common broadcast channel BC. Servers that belong to the write quorum in rows b and $b+1$ also check if object V already exists, and if so whether the new timestamp is greater than what they have stored locally. If the new timestamp is greater, then a message declaring this result is also broadcasted along with the hash and the timestamp. Each server in a write quorum checks if the write quorums in all the rows sent the same timestamp, and for each row a write quorum of servers sent the same hash, and also if at least $b+1$ servers from rows b and $b+1$ certified that the new timestamp is indeed the highest. If all these checks pass, the servers in the write quorum commit the write. The write quorum of servers in rows b and $b+1$ store the hashes of the shares of all the rows in the hash vector $H_{\mathrm{V}}[\,]$. The new timestamp of the data object, TS_{V},

```
Read Protocol for data object $V$
1. Retrieve a read quorum of responses ($<$ server id $i$, row id $r, s_{\mathrm{Vr}} >$) from each row $r$, $1 \leq r \leq (b-1)$,
      and from rows $b$ and $b+1$, responses ($<$ server id $i$, row id $r, s_{\mathrm{Vr}}, H_{\mathrm{V}}[\,], TS_{\mathrm{V}} >$).
         Overhead: $(b+1)RQ_{\mathrm{r}}$ messages, $((b-1)l+(l+(b+1)160)2)RQ_{\mathrm{r}}$ bits
2. Choose the $H_{\mathrm{V}}[\,]$ with the highest timestamp and seconded by at least $b+1$ servers.
      If no such $H_{\mathrm{V}}[\,]$ exists, return null.
3. For each row, choose the share whose hash matches that stored in the chosen hash vector $H_{\mathrm{V}}[\,]$.
      If no such share exists, return null.
4. From the $b+1$ shares thus determined, derive the value of data object $V$. Overhead: 0 exps, b+1 mults

Write Protocol for data object $V$
Part A: If object $V$ does not exist, or if $V$ exists and all shares need to be over-written:
      1. If $V$ already exists:
            1a. If (the current timestamp of $V$ is not known)
                  request the timestamp of $V$ from a read quorum of servers in rows $b$ and $b+1$.
            1b. Choose a timestamp $TS_{\mathrm{V,new}}$ greater than the highest timestamp returned by
                  at least $b+1$ servers and not previously chosen by this client.
      else
            1c. Initialize the timestamp of the object, $TS_{\mathrm{V,new}}$, to 0.
      2. Create $b+1$ shares of data object $V$ using a $(b+1)$-threshold Shamir's scheme.
            Overhead: 0 exps, b(b+1) mults
      3. For each row $r$, $1 \leq r \leq (b+1)$, write $(TS_{\mathrm{V,new}}, s_{\mathrm{V,r}})$ to a write quorum in row $b+1$;
            Overhead: $WQ_{\mathrm{r}}$ messages, $lWQ_{\mathrm{r}}$ bits

Part B: If object $V$ already exists, and an update needs to be done:
      1a. If (the current timestamp of $V$ is not known)
            request the timestamp of $V$ from a read quorum of servers in rows $b$ and $b+1$.
      1b. If $(v, s_1, ..., s_{\mathrm{b}}$ are not known), execute a read on $V$.
      2. Determine share $s'_{\mathrm{V,b+1}}$ using Equation 4. Overhead: 0 exps, (b+1) mults
      3. Choose a timestamp for $V$, $TS_{\mathrm{V,new}}$ greater than the highest timestamp returned (in step (1a))
            by at least $b+1$ servers, and not previously chosen by this client.
      4. Send $(TS_{\mathrm{V,new}}, s'_{\mathrm{V,b+1}})$ to a write quorum in row $b+1$; Overhead: $WQ_{\mathrm{r}}$ messages, $lWQ_{\mathrm{r}}$ bits
```

Figure 3: Read and Write Protocols for Method GridSharing

is stored along with the hash vector at the write quorums in only these two rows of servers. Without loss of generality, it is assumed that the write quorums in rows b and $b+1$ store the timestamp and the hash vector for V. Servers that were not part of the write quorum delete any old shares and associated data (such as the hash vector and the timestamp) of V that they might have. Thus, at the end of the protocol, only servers that were part of the write quorum have shares, and in case of rows b and $b+1$, the hash vector and the new timestamp. Servers in rows b and $b+1$ that were not part of the write quorum no longer have the hash vector and timestamp for data object V. The message and communication complexities on the broadcast channel BC during writes are therefore $(b+1)|WQ_{\mathrm{r}}|$ messages and $(b+1)|WQ_{\mathrm{r}}|160$ bits respectively.

To update the data object V, the client writes to a write quorum of servers in row $b+1$ the new share and timestamp. Servers that belong to the write quorum in row $b+1$ broadcast on the common broadcast channel BC the hash of the new share and the timestamp, and also if the new timestamp is higher than what they have stored locally. The write quorum of servers in row b that have the hash vector and timestamp for V check if the new timestamp broadcast by servers in row $b+1$ is higher than what it has stored locally, and broadcast a message containing the result. If at least $b+1$ servers from the write quorums in row b and $b+1$ find the new timestamp to be higher than what they have stored locally and not more than b servers find the timestamp to be less than what they have stored locally, and the write quorum in row $b+1$ also agree on the same share and the new timestamp (via broadcast of the timestamp and the hash of the share), then the write quorum in row $b+1$ updates their shares and the timestamp. Servers in rows b and $b+1$ that have the hash vector and timestamp of V update these with the new timestamp and the hash of the new share broadcasted during the protocol. Servers in row $b+1$ that are not part of the write quorum delete their local shares of V, but retain any associated data such as the timestamp and the hash vector for object V. This is because some servers in the write quorum in row $b+1$ may not be aware of the hash vector of data object V. These servers will learn the hash vector by the end of the subsequent share renewal phase. The message and communication complexities on the broadcast channel BC during updates are therefore $2|WQ_{\mathrm{r}}|$ messages and $|WQ_{\mathrm{r}}|160$ bits respectively.

Also, thanks to the common broadcast channel, all servers that hold a share of V are aware of the write quorums used to store V in each of the rows. During updates, servers in row $b+1$ (in case of data object V) are aware of the write quorums in rows b and $b+1$ because they participate in the consistency protocol. Servers in row $b+1$ and the write quorum in row b also broadcast the quorum identities in other rows that hold a share of V so that the new write quorum in row $b+1$ can learn this information. This information is needed for efficient share renewal.

The read / write protocols implement safe variable semantics, and satisfy the following theorems.

Theorem 2 *A read on $H_{\mathrm{V}}[\,]$ that is concurrent with no writes returns the $H_{\mathrm{V}}[\,]$ generated in the last write operation, in some serialization of all earlier write operations.*

Proof: When the data object V is created successfully, the hash vector maintained at write quorums in rows b and $b+1$ store the hashes of the shares broadcast by the write quorums in each row. During subsequent successful updates to V, the hash vector and the timestamp is updated at write quorums in rows b and $b+1$ with the hash of the share and the timestamp broadcast by a write quorum of servers in row $b+1$. The update is successful only if not more than b servers claim that the timestamp is not the highest, and at least $b+1$ servers state that the timestamp is greater than what they have locally. These two conditions guarantee that a client update with the highest timestamp will be accepted, provided that any two write quorums intersect each other in at least $b+1$ servers. Our choice of $\lceil \frac{c+b+1}{2} \rceil$ for the write quorum size in each row satisfies this requirement.

During reads, a response seconded by more than b servers indicates that the response is legitimate. The response with the highest timestamp and the same hash vector will be the values written in the last write. The read and write quorums must therefore overlap in at least $2b+1$ servers. The client reads the hash vector and the timestamp from a read quorum of servers in rows b and $b+1$. Since a read quorum size of $\lfloor \frac{c+b+1}{2} \rfloor$ and a write quorum size of $\lceil \frac{c+b+1}{2} \rceil$ are used for each row, the minimum overlap for two rows will be $2b+2$, which guarantees that the hash vector read will be the latest. ∎

Theorem 3 *A read operation that is concurrent with no write operations returns the value written in the last write operation, in some serialization of all earlier write operations.*

Proof: The hash vector $H_{\mathrm{V}}[\,]$ gets updated at a write quorum of servers in rows b and $b+1$ during successful updates. From Theorem 2, the hash vector read will be the latest. Since the read and write quorums in each row intersect in at least $b+1$ servers, and only a maximum of b servers can be faulty, at least one server in each row will return the most recent share. Since the hash vector is read along with the shares in the same request, and the hash function used to compute $H_{\mathrm{V}}[\,]$ is assumed to be collision free, the share corresponding to the hash stored in $H_{\mathrm{V}}[\,]$ can be identified. Reconstructing the data from the shares verified by the latest hash vector guarantees that the reconstructed data is also the latest. ∎

```
**Each server $S_i \in WQ_{Vr}$, the last write quorum of object $V$ in row $r$, $1 \le r \le b+1$, executes:**
1. If $S_i$ has the smallest *id* in $WQ_{Vr}$,
      pick $b$ random numbers $(\delta_{r1}...\delta_{rb})$ from $Z_p$ to define the polynomial $\delta_r(x) = \delta_{r1}x + \delta_{r2}x^2 + ... + \delta_{rb}x^b$,
      and broadcast to all servers in $WQ_{Vr}$ the message $ENC_{rr}(\delta_{r1}, ..., \delta_{rb})$. *Overhead: 1 message, bl bits*
2. Compute subshares $u_{rl}$ for all rows $l$, $u_{rl} = \delta_r(l)(\text{mod } p)$. *Overhead: 0 exps, b(b+1) mults*
3. Send $(r, l, ENC_{rl}(u_{rl}))$ to $S_j \in WQ_{Vl}$, $1 \le l \le (b+1)$, $l \ne r$. *Overhead: b messages, bl bits*
4. If $S_i \in WQ_{Vb}$ or $S_i \in WQ_{V,b+1}$, also send $TS_V$, the timestamp of $V$.
5. If $S_i \in$ row $b$ or $S_i \in$ row $b+1$, and $WQ_{Vb}$ and $WQ_{V,b+1}$ have a timestamp different from
      what $S_i$ has, then delete local share of $V$ and all associated data such as the timestamp and
      the hash vector of $V$, and exit the protocol.
6. Execute $s_{Vr,new} = s_{Vr,old} + \sum_{l=1}^{b+1} u_{lr}(\text{mod } p)$,
      where $u_{lr}$ is the subshare generated by at least $|WQ_{Vl}| - b$ servers in row $l$ for row $r$.
7. Broadcast the hash of the new share to servers in rows $b$ and $b+1$. *Overhead: 1 message, 160 bits*
8. If $S_i \in WQ_{Vb}$ or $WQ_{V,b+1}$, update $H_V[r]$ with the hash returned by at least $|WQ_{Vr}| - b$ servers in row $r$.
```

Figure 4: The Share Renewal Protocol for Data Object V in Method GridSharing

6.2.3 Share Renewal

Since servers delete old shares at the time of client writes itself, only servers that currently have a share of the data object V participate in the share renewal of V. First, servers in each row agree upon the polynomial to be used for generating shares of "0." This can be achieved by having, in each row, the server with the lowest *id* generate the polynomial to be used by only the servers in the same row. Then each server that has a share of the object V generates a subshare of "0" for every row and broadcasts them securely to servers of the respective rows. The subshares generated by a quorum of servers in each row are used.

It is assumed that servers in a row maintain a secret key that can be used to encrypt messages sent amongst themselves (such as the polynomial coefficients to be used on generating shares of "0"). It is also assumed that any two rows share a symmetric key between them for encryption purposes. Thus, a subshare broadcasted by a server in row r for servers in row l can be decoded only by servers in rows r and l.

Once the share renewal process is complete, each server broadcasts the new hash of its share to all the servers in the b^{th} and $(b+1)^{th}$ rows. For each server in these two rows, if the server already has the hash for a row, then it is updated with the hash sent by a write quorum of servers in that row.

We thus save on computational overhead and instead increase the message complexity of the share renewal protocol. The complete protocol is shown in Figure 4. In the given pseudocode, WQ_{Vr} denotes the write quorum of servers in row r that have a share of data object V.

6.2.4 Evaluation Of Performance Metrics

1. **Availability:** Assume each server can fail with probability p, and this probability is independent of other server failures. If there are c servers in each row, then the availability of a read quorum is given by

$$\left(\sum_{f=0}^{c-|RQ_r|} \binom{c}{f} p^f (1-p)^{c-f} \right)^{b+1}, \text{ where } |RQ_r| = \lfloor \frac{c+b+1}{2} \rfloor$$

The availability of a write quorum for the first write to a data object is given by

$$\left(\sum_{f=0}^{c-|WQ_r|} \binom{c}{f} p^f (1-p)^{c-f} \right)^{b+1}, \text{ where } |WQ_r| = \lceil \frac{c+b+1}{2} \rceil$$

The availability of a write quorum for subsequent writes to a data object is

$$\sum_{\mathrm{f}=0}^{\mathrm{c}-|\mathrm{WQ_r}|} \begin{pmatrix} c \\ f \end{pmatrix} p^{\mathrm{f}}(1-p)^{\mathrm{c-f}}, \text{ where } |WQ_{\mathrm{r}}| = \lceil \frac{c+b+1}{2} \rceil$$

2. **Minimum number of servers:** required to tolerate b Byzantine server faults is $(b+1)(3b+1)$.

3. **Size of read and write quorums:** For a given fault threshold b and number of servers c in each row, the read quorum size is given by $(b+1)|RQ_{\mathrm{r}}|$ and the write (update) quorum size is given by $|WQ_{\mathrm{r}}|$, where $|RQ_{\mathrm{r}}| = \lfloor \frac{c+b+1}{2} \rfloor$ and $|WQ_{\mathrm{r}}| = \lceil \frac{c+b+1}{2} \rceil$.

4. **Storage blowup factor:** The size of a share is the same as that of the secret. The size of each hash is 160 bits, assuming the Secure Hash Algorithm (SHA) is used. Since the hashes of $b+1$ shares are stored, the hash vector takes $(b+1)160$ bits of storage. The probability that a certain row of servers is among the two rows chosen to store the hash vector is $\begin{pmatrix} b \\ 1 \end{pmatrix} / \begin{pmatrix} b+1 \\ 2 \end{pmatrix} = \frac{2}{b+1}$. Hence, a row of servers contains $\frac{2}{b+1}$ of hash vectors of all the data objects stored in the system. In addition, at any given time, only a write quorum of servers per row contains the data object, with older shares and hash vectors getting deleted during subsequent writes. The storage blowup factor is therefore $(1 + ((b+1)\frac{160}{l})\frac{2}{b+1})\frac{|WQ_{\mathrm{r}}|}{c}$, where l is the number of bits used to hold a share.

5. **Load:** Let r denote the percentage of read requests and w denote the percentage of write requests generated by the client ($r + w = 1$). The probability that a server participates in a read is given by $\frac{|RQ_{\mathrm{r}}|}{c}$. The probability that a certain row participates in a write operation is given by $\frac{1}{b+1}$, with the probability that a certain server within a chosen row participates in a write given by $\frac{|WQ_{\mathrm{r}}|}{c}$.

   The load is therefore given by $\frac{r}{r+2w}\frac{|RQ_{\mathrm{r}}|}{c} + \frac{w}{r+2w}\frac{2}{b+1}\frac{|RQ_{\mathrm{r}}|}{c} + \frac{w}{r+2w}\frac{1}{b+1}\frac{|WQ_{\mathrm{r}}|}{c}$. The second term is due to timestamp reads.

6. **Confidentiality:** If a server in each row is compromised, an adversary can learn the data. The confidentiality is therefore the probability that there is at least one row of fault-free servers. Assuming the probability that a server can be compromised is p and is independent of other server failures, the confidentiality of the system is given by

$$\sum_{\mathrm{r}=0}^{\mathrm{b}} \begin{pmatrix} b+1 \\ r \end{pmatrix} P^{\mathrm{r}}(1-P)^{\mathrm{b+1-r}}$$

   where P is the probability that at least one server in a given row is compromised, which is $(1-(1-p)^{\mathrm{c}})$.

7. **Computation overhead:** When creating a data object or when overwriting all the shares of an object, referred to as Fresh Write (FW), a client has to generate all the shares of the new data object. Since there is an overhead of b multiplications in the creation of one share and a total of $b+1$ shares are generated, the overhead will be $b(b+1)$ multiplication operations. To do an update (UD) of data object V, where say only the share stored in the $(b+1)^{\mathrm{th}}$ row is updated, a client needs to be aware only of this share, $s_{\mathrm{V,b+1}}$, and the previous value of V. The difference between the new and old values of V multiplied by the inverse of $m_{\mathrm{b+1}}$ (see Equation 4) gives the value that must be added to $s_{\mathrm{V,b+1}}$ to obtain the new share $s_{\mathrm{V',b+1}}$. The client therefore performs only one multiplication operation and no exponentiation operation during writes.

   Servers do not incur any substantial computation overhead during writes (FW) and updates (UD).

   During client reads, servers do not incur any substantial computation overhead. Clients need to recover the secret from the $b+1$ shares, which would involve $b+1$ multiplications (Eqn 3).

A server participating in the share renewal protocol performs b multiplication operations for each of the $b+1$ rows in order to compute the subshares (step (2) of the protocol). No exponentiation operations are required. The computation overhead during share renewal is therefore $b(b+1)$ multiplication operations per data object.

8. **Message and communication complexities:** Denote the length of the prime p by l bits. When evaluating Method RegularSharing, l was used to denote the bit lengths of primes p and q as the two bit lengths are expected to be nearly the same.

   For a fresh write (FW), a client contacts a write quorum of servers in each of the $b+1$ rows giving a total of $(b+1)|WQ_\mathrm{r}|$ messages. Each of these messages is approximately l bits long, giving a total communication complexity of $l(b+1)|WQ_\mathrm{r}|$ bits at the client. During an update (UD), a client sends a modified share and the new timestamp to a write quorum of servers in one row. The message complexity is therefore $|WQ_\mathrm{r}|$ messages. The communication complexity is $l|WQ_\mathrm{r}|$ bits.

   As pointed out in Section 6.2.2, for a fresh write (FW), the message and communication complexities on the broadcast channel BC are $(b+1)|WQ_\mathrm{r}|$ messages and $(b+1)|WQ_\mathrm{r}|160$ bits respectively. During updates (UD), the message and communication complexities on the broadcast channel BC are $2|WQ_\mathrm{r}|$ messages and $160|WQ_\mathrm{r}|$ bits respectively.

   During reads, a read quorum of servers in each row send shares, and a read quorum of servers in two rows send the hash vector. The message complexity is therefore $(b+1)|RQ_\mathrm{r}|$ messages, while the communication complexity is $((b+1)l + 2(b+1)160)|RQ_\mathrm{r}|$ bits.

   For share renewal, referring to the pseudocode in Figure 4, in step (1), one message is sent in each row, with the message length being bl bits. In step (3), a total of $(b+1)(|WQ_\mathrm{r}|b)$ messages are sent, with the length of each message being l bits. In step (7), a total of $(b+1)|WQ_\mathrm{r}|$ messages are sent, with the length of each message being 160 bits.

   The total message complexity of the share renewal protocol is thus $b+1+b(b+1)|WQ_\mathrm{r}|+(b+1)|WQ_\mathrm{r}|$ messages, and the communication complexity is $(b+1)bl + (b+1)bl|WQ_\mathrm{r}| + (b+1)|WQ_\mathrm{r}|160$ bits.

7 Comparison and Discussion

In this section, we seek to bring out the benefits offered by integrating quorum systems and proactive secret sharing, and thus prove that proactive cryptosystems can be made suitable for generic data storage purposes. We compare our two proposed methods, Method RegularSharing and Method GridSharing, against Byzantine quorum systems (replication only) and a slighty-modified Herzberg's proactive secret sharing scheme.

In the pure replication scheme using Byzantine quorum systems, it is assumed that a protocol is run during writes to ensure that all servers in a write quorum commit the same data object. To achieve this, each server in a write quorum broadcasts the hash of the data object it received, and if a write quorum of servers broadcast the same hash and the same highest timestamp, the write is committed.

In secret sharing schemes, each server in the system gets one share, and any $b+1$ shares are required to recover the secret. In order to realize this in practical systems, we have the write and read quorum sizes are given by $(n-b)$ servers and $(3b+1)$ servers respectively, where n denotes the total number of servers in the system and b denotes the fault threshold. These quorum sizes are the maximum and minimum possible in an asynchronous environment where Byzantine failures are possible. With only the quorum sizes being different, this slighty-modified secret sharing scheme and Method RegularSharing are identical.

Table 1 compares these methods in terms of different performance metrics when the number of servers in the system is 44 and the fault threshold is 3. In calculating the availabilities, it is assumed that a server can be compromised or crashed with a probability of 0.1. In confidentiality calculations, the probability that a server is compromised in *one time period* is assumed to be 0.01. In calculating load, it is assumed that read requests constitute 90% of the requests, while the remaining requests are writes. The number of bits used to store

	Pure Quorum Systems	**Pure Secret Sharing**	**Method RegularSharing**	**Method GridSharing**
Write quorum size	26	41	26	8 (updates (UD)) 32 (fresh writes (FW))
Read quorum size	25	10	25	28
Availability of write quorums	7 nines	0.347	7 nines	0.981 (UD) 0.928 (FW)
Availability of read quorums	8 nines	17 nines	8 nines	0.989
Confidentiality	$9.7\text{x}10^{-3}$ (in lifetime of secret)	3 nines (per time period)	3 nines (per time period)	nearly 4 nines (per time period)
Storage Blowup	0.59	4.66	2.95	1.18
Load	0.57	0.29	0.57	0.57
Comp. overhead on clients (writes)	0 exps, 0 mults	4 exps, 123 mults	4 exps, 78 mults	0 exp, 1 mults (UD) 0 exp, 12 mults (FW)
Comp. overhead on servers (writes)	0 exps, 0 mults	4 exps, 3 mults	4 exps, 3 mults	0 exp, 0 mults (FW and UD)
Msg. complexity at clients (writes)	26	41	26	8 (UD) 32 (FW)
Msg. complexity at servers (writes)	26	41	26	16 (UD) 32 (FW)
Comm. complexity at clients (writes)	1.63 KB	12.81 KB	8.13 KB	0.5 KB (UD) 2 KB (FW)
Comm. complexity at servers (writes)	0.51 KB	0.8 KB	0.51 KB	0.16 KB (UD) 0.63 KB (FW)
Comp. overhead (share renewal)	—	163 exps, 326 mults	103 exps, 206 mults	0 exp, 12 mults
Msg. complexity (share renewal)	—	82	52	132
Comm. complexity (share renewal)	—	110.2 KB	45.5 KB	7.38 KB
Comp. overhead on clients (reads)	0 exps, 0 mults	16 exps, 16 mults	16 exps, 16 mults	0 exp, 4 mults
Msg. complexity at clients (reads)	25	10	25	28
Comm. complexity at clients (reads)	1.56 KB	3.13 KB	7.81 KB	2.84 KB
Min. # servers reqd. for $b = 3$	13	13	13	40

Table 1: **Comparison between Pure Quorum Systems, Pure Secret Sharing, Method RegularSharing, and Method GridSharing for $n = 44$ and $b = 3$**

shares and witnesses is assumed to be $l = 512$. The computation, communication, and message complexities are measured on a per data object basis. For Method GridSharing, two values are given for parameters that concern with writes. The values tagged by the letters UD denote the measure when the client does an update by writing to only one row of servers. The value tagged by the letters FW denote the measure when the client creates a data object or decides to write over all the shares of the object, which is required in the first write after changes have been made to the access control list.

Also, the size of the read and write quorums quoted in the table for Method GridSharing is the sum of the quorums used in the required number of rows. In the text so far, "quorums" have been used to refer to the groups of servers in a certain row.

The advantages and disadvantages of pure quorum systems and pure secret sharing can be clearly seen from the table. Since in pure secret sharing writes take place to almost all the servers in the system, the availability of a write quorum is very small. The availability of a read quorum, though, is very high because only $3b + 1$ servers need to be queried. This imbalance is eliminated in pure quorum systems, where the read and write quorum sizes are made almost equal so that their availabilities are equally high. Quorum systems on the other hand provide practically no confidentiality. Compromise of a single server can leak data to an adversary. The adversary has the entire lifetime of the secret to try and compromise a single server. The pure secret sharing technique achieves a stronger confidentiality by requiring an adversary to compromise $b + 1$ servers *in the same time period.* Proactive secret sharing techniques, therefore, provide high confidentiality, while quorum systems provide very high availability.

The computation, communication, and message complexities during share renewal are quite high for the pure secret sharing approach. Since we would like to design a generic data storage service, the amount of data that will be stored will be in the order of giga- or terabytes, while the data object size we have assumed is only 64 bytes. The share renewal for all objects at time period boundaries will take a long time to complete and impose a considerable amount of load on the servers. The share renewal process is therefore not scalable with the amount of data stored in case of the pure secret sharing technique.

Since generic-data storage techniques that achieve high availabilities for the read and write quorums and minimum costs to provide high long-term confidentiality is our primary goal, combining the proactive secret sharing and quorum systems would seem to be a strong contender to meet our needs.

Our proposed solutions, Method RegularSharing and Method GridSharing, have the answers. Method RegularSharing is very similar to pure quorum systems, with each server in a write quorum getting a distinct share instead of the data object being replicated throughout the write quorum. Thus, Method RegularSharing combines the positive aspects of pure quorum systems and pure secret sharing. Its availability is the same as that of pure quorum systems, while its confidentiality is the same as pure secret sharing. This method further improves upon the share renewal protocol, especially with the communication complexity of the share renewal protocol, and the message complexity during writes compared to pure secret sharing. This can be attributed to the write and the share renewal protocols being restricted to a smaller write quorum for a given object compared to the pure secret sharing approach. For the same reason, the storage blowup factor is reduced for Method RegularSharing when compared against pure secret sharing.

Method RegularSharing still suffers from heavy computation overheads during the share renewal protocol. Method GridSharing was designed to ease the share renewal protocol, while retaining as much as possible the benefits provided by the pure quorum systems approach and the pure secret sharing approach. This method replicates shares to some extent and does not use Feldman's VSS for verifiability during share renewal, which is the prime culprit in causing the computation overhead in Method RegularSharing and in the pure secret sharing approach. The computation overhead during share renewal in Method GridSharing is reduced to performing only 12 multiplication operations per data object and no exponentiation operations. Even though Method GridSharing has a higher message complexity than Method RegularSharing and the pure secret sharing approch during share renewal, the communication complexity is relatively very small. Computation overheads are usually higher than the message delays on dedicated communication channels. Thus, Method GridSharing implements an efficient practical share renewal protocol.

When objects are modified in Method GridSharing, if clients cache the old copy of the secret and the share in one row, then a simple update (UD) operation can update the servers with the modified secret or data object. Therefore, some extra storage requirements are required at the client end. If a client decides to write (FW) all the shares of the data object (and hence avoid the need for client-side caching), then it will incur a relatively higher overhead in communicating to the data servers.

Regardless of whether a client does an update (UD) or a fresh write (FW), the overheads during writes on servers and clients are less than the pure secret sharing approach. The overheads are also less compared to Method RegularSharing, except for the message complexities at the client and servers during writes when a full write takes place. The increase in overhead is small. The communication complexity at the clients (both for FW and UD) and the communication complexity at the servers during UD is much smaller than Method RegularSharing. The communication complexity during FW is only marginally higher than Method RegularSharing.

The confidentiality provided by Method GridSharing is the same as that of pure secret sharing approach and Method RegularSharing. Though Method GridSharing does not have read and write quorum availabilities as high as that of pure quorum systems and Method RegularSharing, its availabilities may still be acceptable. But for a given fault threshold b, Method GridSharing requires more servers that the other three methods. In other words, for a given number of servers, Method GridSharing is able to tolerate fewer compromised servers.

Method RegularSharing and Method GridSharing are suitable in two different kinds of environments. Method RegularSharing is suitable in environments where clients are powerful and write data often. Due to data being written often, there may not be the need to do share renewal often, and also the write quorum availability needs to be high. Method GridSharing is suitable in environments where clients do not have enough computation power and do not modify data often. When they do modify data, it is more of making changes than creation of new data objects. An example would be using the storage service for backups, and where the client periodically updates the backups to make them current. Since most data are expected to be static or archival in such environments, it is important to do share renewal often.

8 Future Work

There is ample scope for future work. The assumption of a zero-delay broadcast channel for the purposes of share renewal and verifiable secret sharing may not always be realistic, and warrants the use of asynchronous proactive secret sharing schemes [30, 29]. In an asynchronous environment, the message and communication complexities will be much higher, and even the notion of proactive security will require a new definition. There can be no common clock to alert the servers to start share renewal at the same time. Servers will need to engage in a Byzantine agreement protocol for verifiable secret sharing and share renewal, and it remains to be seen where the line between practical and theoretical interests can be drawn.

Even when a synchronous broadcast channel is assumed, improved share renewal protocols are needed. Method GridSharing solves this while sacrificing availability to some extent. Since secret sharing itself provides secrecy, it may be possible to encrypt shares using a very weak encryption algorithm, and then do share renewal by simply changing the key and having servers independently re-encrypt their shares, while never recovering the original shares in the process. This and other methods may be possible and needs to be explored. Lastly, Feldman's verifiable secret sharing scheme is too computationally intensive for generic data storage uses. Method GridSharing solves this by using voting. There is room for novel solutions in this area as well.

References

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

[2] M. Herlihy and J. D. Tygar, "How to make replicated data secure," in *CRYPTO*, pp. 379–391, 1987.

[3] A. Adya and et al., "Farsite: Federated, available, and reliable storage for an imcompletely trusted environment," *Proc. of the* 5th *Symp. on Operating systems design and implementation*, Dec. 2002.

[4] "Mojonation." http://www.mojonation.net.

[5] Y. Chen and et al., "A prototype implementation of archival intermemory," *Proc. of the* 4th *ACM Intl. Conf. on Digital Libraries*, pp. 28–37, Aug. 1999.

[6] J. Kubiaotowicz and et al., "Oceanstore: An architecture for globalscale persistent storage," *Proc. of ASPLOS IX*, pp. 190–201, Nov. 2000.

[7] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant web publishing system," *Proc. of 9th Usenix Security Symposium*, pp. 59–72, 2000.

[8] R. J. Anderson, "The eternity service," *Proc. of First International Conference on Theory and Application of Cryptography (Pragocrypt)*, 1996.

[9] A. Iyengar, R. Cahn, C. Jutla, and J. Garay, "Design and implementation of a secure distributed data repository," in *Proc. 14th IFIP International Information Security Conference*, pp. 123–135, 1998.

[10] "Pasis." http://www.pdl.cmu.edu/Pasis.

[11] R. Dingledine, M. J. Freedman, and D. Molnar, "The free haven project: Distributed anonymous storage service," *Proc. of Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[12] Y. Deswarte, L. Blain, and J. C. Fabre, "Intrusion tolerance in distributed computing systems," in *Proc. 14th IEEE Symposium on Security and Privacy*, pp. 110–121, 1991.

[13] G. R. Blakley, "Safeguarding cryptographic keys," *Proc. of the National Computer Conference*, pp. 313–317, 1979.

[14] M. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM*, vol. 38, pp. 335–348, 1989.

[15] L. Kong, A. Subbiah, M. Ahamad, and D. Blough, "A reconfigurable byzantine quorum approach for the agile store," in *Proc. of the 22nd International Symp. on Reliable Distributed Systems*, pp. 219–228, 2003.

[16] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. N. Wright, "Dynamic byzantine quorum systems," in *International Conference on Dependable Systems and Networks*, vol. 1, 2000.

[17] Y. Desmedt and S. Jajodia, "Redistributing secret shares to new access structures and its applications," *Technical Report ISSE TR-97-01, George Mason University*, 1997.

[18] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the 7th SOSP*, pp. 150–162, 1979.

[19] G. Agrawal and P. Jalote, "Coding based replication schemes for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 240–251, Mar 1995.

[20] G. Agrawal, "Availability of coding based replication schemes," in *Symposium on Reliable Distributed Systems*, pp. 103–110, 1992.

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

[22] J. Jiang, "Secret sharing via nondominated write-read coteries," *Proc. of the* 13th *Intl. Conf. on Parallel and Distributed computing systems*, pp. 561–565, Aug. 2001.

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

[24] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, “Responsive security for stored data,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, Sept 2003.

[25] R. Ostrovsky and M. Yung, “How to withstand mobile virus attacks,” *Proceedings of the* 10^{th} *Symposium on the Principles of Distributed Computing*, 1991.

[26] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, “Proactive secret sharing or: How to cope with perpetual leakage,” in *CRYPTO*, pp. 339–352, 1995.

[27] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, “Verifiable secret sharing and achieving simultaneity in the presence of faults,” in *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pp. 383–395, 1985.

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

[29] L. Zhou, “Towards fault-tolerant and secure on-line services,” *PhD Thesis, Cornell University*, 2001.

[30] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, “Asynchronous verifiable secret sharing and proactive cryptosystems,” *Proc. of the* 9^{th} *ACM Conference on Computer and Communications Security*, Aug. 2002.

[31] D. Malkhi and M. Reiter, “Byzantine quorum systems,” *Distributed Computing*, vol. 11, no. 4, 1998.

[32] L. Lamport, “On interprocess communication, part 1: Basic formalism,” *Distributed Computing*, vol. 1, pp. 77–85, 1986.

[33] J. Wylie and et al, “Selecting the right data distribution scheme for a survivable storage system,” *Technical Report CMU-CS-01-120, Carnegie Mellon University*, 2001.