Responsive Security for Stored Data *

Subramanian Lakshmanan Mustaque Ahamad H.Venkateswaran

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{subbu,mustaq,venkat}@cc.gatech.edu

Abstract

We present the design of a distributed store that offers various levels of security guarantees while tolerating a limited number of nodes that are compromised by an adversary. The store uses secret sharing schemes to offer security guarantees namely availability, confidentiality and integrity. However, a pure secret sharing scheme could suffer from performance problems and high access costs. We integrate secret sharing with replication for better performance and to keep access costs low. The tradeoffs involved between availability and access cost on one hand and confidentiality and integrity on the other are analyzed. Our system differs from traditional approaches such as state machine or quorum based replication that have been developed to tolerate Byzantine failures. Unlike such systems, we augment replication with secret sharing and offer weaker consistency guarantees. We demonstrate that such a hybrid scheme offers additional flexibility that is not possible with replication alone.

Keywords: Security, Byzantine fault tolerance, replication, secret sharing, availability, confidentiality, data integrity, distributed storage service.

1. Introduction

Many applications need to store long-term data and retrieve it for later use. These applications range from single user applications storing personal information to multiuser collaborative applications that allow users to share stored data. Furthermore, these applications may run on computers that range from desktops to mobile and hand held devices. For example, the **Aware Home** [1] that has been built at the Georgia Institute of Technology is an information rich environment where a number of devices capture information about the residents and their activities. Such information, which could be of sensitive nature, is stored and accessed by a number of applications. Several characteristics of computers that execute such applications make them unsuitable for storing sensitive information. First, the devices may be resource poor and may not be able to store long-term data. Second, they can be easily stolen or compromised and hence cannot be trusted with long term storage of data that has confidentiality and integrity requirements. Third, when data size becomes large, storage management is expensive and prone to errors. As ubiquitous applications proliferate, the need for services that can store data securely will arise in environments that span the home and community where fewer assumptions can be made about proper system management.

It would be desirable to provide an easy to manage data repository service for applications that store and access sensitive information. The kind of data that would be stored in the data storage service imposes several requirements on its design. First, the data has to be highly available and quickly accessible to distributed clients. Second, access to private or sensitive data should be controlled. In particular, confidentiality and integrity of the stored data should not be compromised. Third, these requirements should be met even when some number of servers in the secure store are compromised by an adversary.

We present the design and analysis of a distributed data store that can meet both security and performance requirements. Our design integrates two well known techniques, namely replication and secret sharing to achieve this goal. Our key contribution is an architecture that provides desirable levels of security guarantees and performance by exploiting the natural tradeoffs possible between the two conflicting goals. We present protocols that allow data to be read and written in a system that replicates data shares, even when some number of nodes can be compromised. To provide additional security we allow periodic share renewal

*This work was supported in part by NSF grants CCR-9988212, ITR-0081276 and CCR-TC-0208655.

when necessary because of the nature of the data. Similarly we use update dissemination among replicas to improve performance. Owing to replication, our store must address consistency of data when it is read and updated. We offer weaker yet useful levels of consistency for stored data. Our store offers two weak forms of consistency, namely Monotonic Read Consistency and Causal Consistency [31, 30]. We present an analysis of the system to demonstrate the tradeoffs that are offered by our design.

The rest of the paper is organized as follows. In the next section, we describe approaches based on replication and secret sharing briefly, and argue why a combination of the two offers a better solution. In section 3, we discuss related work that has been done in the past. In section 4, we describe the architecture of the secure store and discuss the related protocols. We discuss the consistency issue in section 5 and give a modified set of protocols to guarantee two types of data consistency. We do a simple analysis of our system in section 6 and show the tradeoffs permitted by our design for various security and performance requirements. We conclude with a discussion about future work in the last section.

2. Approach

There are two distinct approaches for building a highly available distributed storage service. One is the well known replication technique [6, 7]. Second approach is based on secret sharing schemes [3, 5]. Each of these techniques focuses on optimizing different sets of requirements. While replication enhances availability and increases performance by exploiting data locality and keeping access costs low, schemes based on secret sharing offer better data confidentiality when some number of nodes are compromised.

In a pure replication scheme, the servers are replicas of each other, storing copies of the same set of data items. When such a scheme is used, data has to be encrypted before storing it at servers to offer data confidentiality. A client which does the encryption, would hide the key from the servers, so that a compromised server cannot disclose any information stored at it. When the data so encrypted is shared among a dynamic set of clients, key management becomes an important issue. Whenever a client leaves the set of authorized clients, data has to be re-encrypted using a new key and the new key has to be distributed to the remaining clients. Furthermore, keys have finite lifetime and data has to be periodically re-encrypted using a new key. All replicated copies have to be renewed. If one of the servers is compromised, such a server could retain a copy of the data encrypted with the old key, and content of long-lived data could be leaked over time. Thus, even if data is encrypted with a key that only authorized clients know, storing all the information in encrypted form entirely at any server site would lead to a possibility of information leakage.

A (b, k) secret sharing scheme [3, 10] transforms a data item into k pieces (called data shares or fragments) such that b or fewer shares do not give any information about the data content and any $b + 1$ shares can be used to reconstruct the original data value. This scheme can be used to guarantee both data confidentiality and data availability when the number of compromised nodes is not more than b. Secret sharing schemes offer confidentiality through access control at the servers as opposed to encryption schemes that are based on problems that are hard to compute.

Secret sharing schemes, at the expense of higher communication and computational cost, eliminate the problem of key management. Some secret sharing schemes also allow periodic renewal of shares by the servers without client participation [5]. If the adversary is limited to compromising no more than b nodes in any time-interval of certain length, say T_v units, by doing share renewal at a faster rate, no information would be leaked to an adversary ever. Thus, secret sharing schemes are capable of guaranteeing long term secrecy of data content.

While a pure secret sharing scheme offers better confidentiality, such a scheme would result in high access cost. For example, consider a system of n servers. Let us assume that not more than b servers are compromised. Consider transforming a data item into n shares using a (b, n) secret sharing scheme . Writing such a data item would involve contacting all n servers. Reading would involve contacting a minimum of $b + 1$ servers and upto $2b + 1$ servers when some servers are compromised. When n is very large, write cost could be significantly high even when the number of compromised servers is small.

The cost of write operations could be reduced by allowing a client to write only $2b+1$ data shares and then generating rest of the shares at other servers from the already written ones [5]. Such a generation of each data share involves one or more rounds of $O(b^2)$ messages exchanged between $2b+1$ or more servers. Thus, generating new shares at other servers is a costly and time-consuming process.

Yet another approach is one that combines data replication with key fragmentation. In this scheme, a data item is encrypted and the encrypted version is replicated across servers. The key that is to be shared among clients is fragmented using a secret sharing scheme and distributed across servers. Thus, any client that wants to access a data item would first obtain the key by contacting $b + 1$ servers and then access an encrypted copy of the data item. This is the approach taken by Herlihy et. al. [23]. An extension to this scheme is the scheme proposed by Krawczyk [24] which secret-shares encrypted data using Rabin's information dispersal algorithm [11]. We consider this scheme as one of many secret sharing schemes that our proposed system could use.

In this paper, we propose integrating the two approaches, replication and secret sharing, to meet both performance and security requirements of applications. In our approach, servers are divided into c distinct sets, where c is typically $2b + 1$. Servers in the same set store replicas of the same data shares. Initially a data item is transformed into c shares using a (b, c) secret sharing scheme and one share is written to each of the c sets, choosing one server from each set. A share is then disseminated to other servers in the same set. A client would contact $b + 1$ servers that store distinct shares to retrieve a data item. Access cost as seen by the client is low since a client needs to contact a maximum of only $2b + 1$ servers for reads and writes, even when n is large. Here, we are interested only in access cost as seen by the client and hence ignore the cost involved in disseminating shares to other servers (dissemination can be done asynchronously in the background across nodes that implement the secure store). Higher data locality is possible because of replication of shares which could lead to better performance. Thus, the system is able to offer improved data confidentiality, while offering some of the benefits of replication schemes.

Guaranteeing strong forms of consistency(e.g.,one-copy serializability, sequential consistency or linearizability [27, 29, 2, 8] for replicated data results in expensive protocols that cannot scale (e.g, state machine approach and quorum protocols). Many applications that handle personal data or data that is shared only to a limited extent among multiple users do not need such strong consistency guarantees. Hence we choose to offer weaker consistency levels that are useful to many applications and yet efficient to implement. Our store offers two forms of weak consistency guarantees, namely Monotonic Read Consistency and Causal Consistency.

Pure replication and pure secret sharing schemes are extreme cases of the generalized scheme presented here, when c is set to 1 and n respectively. By increasing c from 1 to n, the system goes from a purely replicated one to a purely secret-shared system. Thus, by increasing c, the system offers better security against compromised servers at the cost of low availability and high access costs . The choice of c depends on the relative importance of security requirements to performance needs. Thus, integration of replication with secret sharing results in the design of a secure store that retains benefits of both these techniques and offers tradeoffs that are not possible with either technique alone.

3. Related work

Replication schemes that tolerate Byzantine faults[1] in a distributed environment have been studied well in the past. Schneider presented a generalized state machine replication approach to fault tolerance [6]. Liskov and Castro gave a practical implementation of the state machine approach for a file system that tolerated Byzantine faults [7]. The state machine approach does not scale well with large number of servers for the reason that each request involves two or more rounds of communication involving all the servers.

Quorum systems are popular for managing replicated data. The Phalanx [2] and Fleet [8] systems were built using a quorum approach to tolerate Byzantine faults. In a quorum scheme, operations are done with sets of servers (quorums) that sufficiently overlap with each other to tolerate some number of malicious failures. Though access cost can be reduced to some extent by weakening the consistency guarantee, it can still be quite high for some class of applications. Alvisi et. al. presented a scheme to dynamically change the threshold value (the number of failures to be tolerated) based on the estimated number of faults perceived [9] and thus avoiding the use of large quorums when the actual number of compromised nodes is small.

Both state machine and quorum based approaches do not offer data confidentiality unless encryption schemes are used. Alternatively, secret sharing schemes that do not use encryption keys were developed to offer data confidentiality even when some number of nodes are compromised. Shamir and Blakley gave simple secret sharing schemes based on polynomial interpolation and intersection of hyper planes, respectively [3, 10]. A number of other schemes have also been developed [11, 13, 24]. Tompa et. al. [12], Feldman [14], Pederson [15] and Krawzcyk [25] have also considered malicious corruption of shares either by servers or by a client.

More recently Herzberg et. al. developed a proactive secret sharing scheme where servers could proactively recover and renew their shares in a distributed manner to protect information against an adversary who can dynamically compromise nodes [5]. Our discussion here uses techniques presented in this paper, in particular, the share renewal and share recovery protocols.

Herlihy and Tygar developed a scheme where data is encrypted and the key is secret-shared [23]. Krawzcyk presented a computationally secure secret sharing scheme combining secret sharing with encryption and Rabin's information dispersal [24]. Naor and Wool presented a scheme in which access control servers are different from storage servers [26]. However, they considered the case of benign server faults and malicious clients.

The PASIS project [16] at CMU very closely relates to our work. PASIS considers various secret sharing and other schemes to store data securely in a data repository. However, PASIS does not consider integration of replication and secret sharing. Fray et. al. proposed an approach similar to ours, fragmentation-scattering [21], where fragments of ci-

[1] A compromised server can behave in arbitrary manner and is assumed to suffer from a Byzantine failure.

pher text are replicated. However replication was achieved by clients broadcasting the fragments. They did not consider fragment dissemination or periodic renewal of fragments. Other related works include data dissemination in benign [17] and malicious environments [18, 19, 20].

Consistency models ranging from strong consistency guarantee like atomicty or one-copy serializability to weaker models for replicated data have been studied well in the past [27, 28, 29]. Weaker consistency models that meet the needs of many applications and yet efficient to implement have been proposed for systems like distributed file systems and shared memory systems. For example, the causal consistency model [30] permits more efficient implementations and can meet the needs of many applications. Although weaker consistency models may be appropriate for some applications, no single model may meet the consistency needs of all applications. Thus, several consistency levels may have to be provided in the same system [31, 32]. The Bayou system best exemplified this approach. Motivated by Bayou-like systems, our system also offers two levels of consistency, namely Monotonic Read Consistency and Causal Consistency. However we differ from Bayou-like systems in that the burden of maintaining consistency is upon clients that must save and use meta-data to decide what values can be accessed without violating consistency.

4. System Architecture and Protocols

Our secure store is implemented by a set of n servers. Clients make read and write requests with subsets of servers. Each request is authenticated and authorized individually by every server. Hence, we assume the presence of appropriate public key infrastructure. Each client and server node has a private key for which the public key is well known. Besides these keys, clients and servers also negotiate symmetric keys periodically to exchange messages. Thus, we assume all communication channels are secure against eavesdropping, modification and replay attacks. Requests are authorized using access control lists at each server, which are updated securely and in a timely fashion by a system administrator, possibly using a separate service.

We assume a Byzantine fault model [4] for servers, where a compromised node can behave arbitrarily. Any compromised node can disclose data shares it holds, corrupt the shares and possibly collude with other compromised nodes. Both our system architecture and protocols are designed to tolerate a certain number of Byzantine faults. This number, denoted by b, is referred to as the threshold value of the system in this paper. For the rest of the section, we assume that a threshold value b has been chosen and that in any time interval of length T_v units, not more than b servers are compromised. We refer to the constant T_v as the vulnerability window.

The set of n servers is arranged logically in a two-dimensional matrix as shown in figure 1. The server in i^{th} row and j^{th} column is denoted by S_{ij}. Number of columns is denoted by c and number of rows r with $rc = n$. We assume that c is at least $b + 1$. A data item is fragmented into c shares using a (b, c) secret sharing scheme and is stored as shares at the servers. For a particular data item, servers along a column store copies of the same data share and each column stores a different share. Both security and performance levels change with values chosen for b and c, as we will see in section 5.

The operation of our secure store is characterized by the following three sets of protocols:

1. Read and write protocols that are used by clients to access the data.
2. A dissemination protocol which is used by the servers to propagate new data shares among themselves.
3. A share renewal protocol that is run periodically to generate new data shares for long-term confidentiality.

Data consistency becomes an issue when data is replicated. We postpone the discussion on data consistency until next section. Our store is capable of offering two kinds of weak consistency guarantees, namely Monotonic Read Consistency and Causal Consistency. For the sake of clarity, the read and write protocols we present in this section do not address consistency requirements.

In this paper, we use a specific verifiable secret sharing scheme due to Feldman [14] for our protocols. The use of this scheme, as discussed in [5], results in (1) easy verification of a share during dissemination, (2) renewal of shares in a purely distributed fashion, and (3) regeneration of lost shares. We could use any secret sharing scheme that has these properties. In the following subsections, we discuss the protocols for reading and writing data, dissemination and share renewal.

4.1. Read and Write Protocols

We do not consider the case of clients being malicious. While malicious clients cannot do any harm to data items for which they do not have access, they can, however, exhaust a server's storage or write garbage to the data items for which they have write access. We rely on detecting malicious clients and using authorization and access control mechanisms to stop malicious clients from doing harm.

Figure 2 shows the write and read protocols when all clients are assumed to be non-malicious. In a write operation, a client fragments a data item into c shares using a (b, c) secret sharing scheme. One-way functions $h(x)$ are

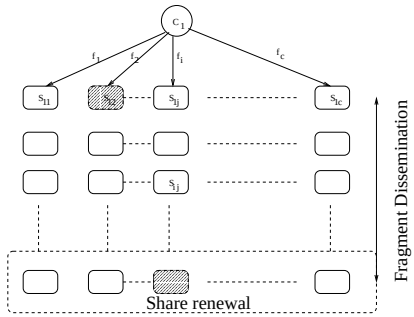

Figure 1. Secure Store Architecture

computed for these shares and concatenated to form a verification string. We discuss issues related to the choice of an one-way function in the following subsections. Verification string is required to let a server know if a share it received in dissemination has been corrupted. Verification string also helps a reading client choose the right set of $b+1$ shares to reconstruct the data. For write, the shares are sent to servers along a row, each receiving a different share along with uid of the data item, timestamp, the verification string and the signature of the whole write. Since some servers that are contacted can be compromised, the write is repeated with different rows until the number l of rows contacted is such that $c - \lfloor b/l \rfloor$ is at least $b+1$. Since maximum number of columns in which all l servers contacted are compromised is $\lfloor b/l \rfloor$, writing to l rows ensures that in each of $b+1$ or more columns, at least one non-malicious server has received the write message.

While reading, a client sends a read request for the object to servers along a randomly chosen row. It collects $b+1$ or more shares corresponding to the same timestamp and reconstructs the data value. Finally it verifies the signature before accepting the value. If the read is not successful, the client contacts additional servers. Variations of this protocol are possible by varying the number and choice of servers a client contacts initially and by varying the way additional servers are contacted.

4.2. Dissemination

Our write protocol may write each share to only one server in a column. To provide better performance and availability, shares written at one set of servers should be disseminated to other servers so that the data is available for access at other servers. Hence, in the secure store, shares are disseminated along columns. Data dissemination for non-malicious environments was studied in [17]. Presence of malicious servers requires a more careful treatment.

Write(x_j, v) by client C_i

1. Let timestamp ts = current clock value concatenated with uid(client).
2. Fragment value v into c shares v_1,v_2,..v_c using a (b, c) secret sharing scheme.
3. Compute one-way function of each of the shares, $h(v_i)$.
4. Form the verification string VS and compute signature
   $VS = h(v_1)|h(v_2)...|h(v_c)$. (concatenation).
   $sig = \{uid(x_j), ts, v\}_{K_{c_i}^{-1}}$, where $\{\}_{K_{c_i}^{-1}}$ denotes signature using private key of the client.
5. Choose a row k.
   for (m = 1 to c){
   send {"write",uid(x_j),ts,v_m,VS,sig} to server S_{km}.
   }
6. Repeat 5 for a different row until $c - \lfloor b/l \rfloor \geq b+1$ where l is the number of rows contacted.

Read(x_j) by client C_i

1. Choose a row k.
   for m = 1 to $2b+1$ {
   send {"read",$uid(x_j)$} to S_{km}.
   }
2. Receive a list of timestamps from each server with the corresponding data shares and verification strings.
3. A timestamp is said to be "good" if it appears in at least $b+1$ replies (lists) and the corresponding verification strings are the same. Let t_r be the highest timestamp among such good timestamps.
4. If there is no good timestamp, repeat from 1 for a different k.
5. Pick shares corresponding to t_r. Pick $b+1$ shares among these that are successfully verified by the verification string. Reconstruct the data value.
6. Check if signature is valid. If valid, return the reconstructed data value. If signature is not valid, repeat from 1 for a different k.

Figure 2. Write and Read protocols

Byzantine nodes can modify the data being disseminated and thus can compromise the availability and integrity of the data. One approach for secure dissemination is to attach an unforgeable signature of the writing client to the data being disseminated. We disseminate data shares rather than the data item itself. Since shares could change over time due to periodic share renewal (dicussed in section 4.3), the client has to recompute the signature of the shares. However, it is a desirable feature to do share renewal without requiring client participation, as we will see in section 4.3.

Secure dissemination schemes without public key signatures have been addressed by others [18, 19, 20]. These schemes require data to be written to at least $b+1$ non-faulty nodes initially before being disseminated. We cannot adopt

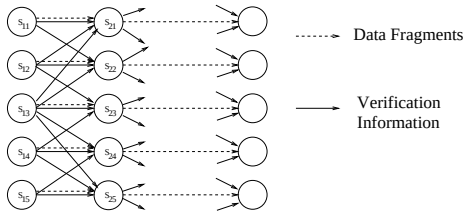

Figure 3. Secure Dissemination

these schemes to disseminate data shares for the reason that each share is written to only one server initially.

We use verification strings to secure the dissemination of data shares. Usual signature verification is replaced by a set of one-way function verifications. Thus, a server verifies a data share f it receives in dissemination by checking it against the one-way function of the share $h(f)$. The verification string, which is a concatenated list of one-way functions of the shares of a data item, should be fully reliable since a corrupted verification string can verify a manipulated share to be correct. Hence, verification string itself is written by the client along with shares and disseminated to other servers securely using any of the secure dissemination schemes [18, 19, 20]. We require that the verification string be disseminated across columns, among all servers. A server accepts a verification string as valid only if $b + 1$ or more servers are known to have accepted the same verification string for a given timestamp. Once a server accepts a list of one-way functions as valid for a timestamp, it accepts the corresponding share by verifying that the one-way function applied on that particular share matches the corresponding part of the verification string. This way, a compromised server cannot modify and disseminate a corrupted share without going undetected, even when colluding with other malicious servers.

Hence, our dissemination protocol works in two parts: (1) dissemination of verification string, (2) dissemination of shares. Figure 4 describes the dissemination protocol. In addition to dissemination of shares and verification string, we add a component to detect corrupted shares and regenerate correct shares. Servers also probabilistically suspect corruption of shares and verify the correctness of shares by pulling verification strings from other servers. Share recovery involves getting secondary shares from $b + 1$ or more other servers, each from a different column, to recover the corrupted or missing share. This scheme is described in [5]. Once the right share is constructed, this share is disseminated along the column. Share recovery is costly and we expect that, if there are only few malicious servers in the system, share recovery would be done only infrequently.

During share renewal, the shares of a date item change, but the new verification string can still be computed securely and reliably, even in the presence of active attackers during the phase of share renewal [5]. Thus, even after share renewal, we can disseminate the new shares securely as we did before share renewal.

Dissemination of verification string

1. During a write operation, client writes VS to all servers it contacts(at least $2b + 1$).
2. A server accepts a VS as valid for a write only if it hears from a client directly or when $b + 1$ other servers have accepted the same VS as valid.
3. A server disseminates a VS to all other servers, both within and across columns.

Dissemination of shares

1. A server disseminates the shares it receives to other servers in the same column. Shares do not have to be verified before being disseminated to other servers.
2. A server accepts a share as valid if the share is successfully verified by a valid VS.
3. Shares can be served to clients even before verification.

Detecting corruption and generating correct shares

1. A server detects corruption if it receives two or more different shares for the same write.
2. A server can probabilistically suspect corruption of the shares it holds.
3. Upon detection or suspicion, a server uses a valid VS to check the validity of the shares. If the server does not have a valid VS, it pulls VS from other servers.
4. If a server finds a share to be corrupted, and does not find the correct share with any other server in its column, it initiates a share recovery protocol with servers from $b + 1$ or more other columns.

Figure 4. Dissemination protocol

4.3. Share Renewal Protocol

We assume that an adversary cannot compromise more than b nodes in any time frame of length T_v . However, this does not prevent an adversary from compromising $b + 1$ or more nodes over a longer period of time and obtain $b + 1$ or more shares to learn the content of a data item. Hence, the shares need to be periodically renewed, in a distributed manner, so that an adversary who obtains b shares before share renewal cannot use them in any manner in future to gain any information, even if he finds additional data shares. The share renewal of a data item is done without the participation of the client that wrote it. This enables share renewal even when the client is offline for an extended time period.

Share-renewal is very expensive and the frequency with which shares of a data item are renewed can be tuned de-

pending on the sensitivity of the stored data item. Shares of data items that lose value over time are renewed less frequently as they age. Also shares of data items that are frequently over-written by clients do not need renewal. Share renewal is done only for those data items that need long term secrecy.

In this paper, we use the share renewal protocol proposed by Herzberg et. al. for Feldman's secret sharing scheme as discussed in [5]. For a given data item, servers belonging to one row initiate a share renewal protocol. At the end of share renewal, the shares are renewed while the data content and other meta-data are retained. The verification string is also updated securely for the new shares. The protocol guarantees that if the number of non-faulty servers is a majority, at the end of the protocol, all non-faulty servers hold valid shares and a copy of the same valid verification string. From then on, the new shares are disseminated as before. No data share is stored at any non-malicious server beyond T_v seconds after its renewal. A share is erased either when a new share arrives or when the share expires. This is critical to guaranteeing confidentiality since share renewal schemes rely on erasing the old shares. Herzberg et. al. [5] assume a secure broadcast channel for share renewal protocol. In our system, we could dedicate a set of servers for this purpose, on a single shared wire, doing share renewal for different data items.

The one-way function used in Feldman's scheme is g^x where g is a primitive element in the field[2] from which values for x are chosen. While this works for our solution, this is costly in terms of storage space required (large verification string) and in terms of computation.

Data items that are over-written frequently by clients and those that do not have strict long-term confidentiality requirements do not need share renewal. For such data items, eliminating share renewal eliminates the need to use an expensive one-way function like g^x for verification string. In such cases, a simple digest function like MD5 could be used for $h(x)$. Data items that need stronger secrecy guarantee would use the more expensive one-way function g^x.

To save on storage space, we use $h(g^x)$ where h is a cryptographic digest like MD5 at the expense of incurring an additional round during share-renewal. Although verification using such expensive one-way functions is computationally intensive, this cost would not be incurred in the dissemination protocol in the common case when most of the servers are not malicious. Only when servers detect corruption of shares, or suspect corruption probabilistically, the verification cost would be incurred. However, reading and writing for such cases would be computationally expensive. So would be share-renewal in the absence of writes. This is a tradeoff clients are offered to decide if a data item should be stored at such a high security level. In place of the function g^x for $h(x)$, we could use any one-way function that satisfies certtain properties as required by the share-renewal protocol. We are currently exploring possible alternative one-way functions that can be used in place of g^x.

[2]For the purpose of this paper, values for data items and data shares are assumed to be chosen from the finite field Z_p, for an appropriate prime p.

5. Consistency

As a consequence of our design and the needs of applications we target, our store offers only weaker levels of consistency for stored data. A variety of relaxed consistency models have been explored in the past that are suitable for different classes of applications. Our system offers two forms of consistency guarantees, namely Monotonic Read Consistency and Causal Consistency. In this section, we define these consistency models, discuss aplications for which these consistency models are suitable and finally give a modified set of read and write protocols to guarantee the required consistency level.

5.1. Consistency Models

We consider the following consistency models that can meet the needs of many of the applications.

1. **Monotonic Read Consistency (MRC):**A client who accesses data items using this level of consistency always sees an increasing set of writes to a an object as time proceeds. More specifically, if a client reads a value v for a data item x_i, at a later time when it reads data item x_i again, it is returned v or a value which is newer than v. A value v' is said to be newer than v if, based on a clock, the write that produced v' is ordered after the write which stored v. For non-shared data that is accessed by a single client, MRC implies that the client will access the most recent copies of its data items. For shared data, future reads of a reader could return more recent values but are not guaranteed to return the latest value of the object.

2. **Causal Consistency (CC):** MRC only ensures that for a given data item, a client never receives values older than the ones read in the past. It does not impose restrictions across related data items. Consider the case when a client writes value v_2 to a data item x_2 based on value v_1 of data item x_1 that it has read. Informally, if another client reads value v_2 for x_2, CC ensures that the client is guaranteed not to read a value for x_1 that is older than v_1. The notion of "older" is precisely defined based on the happens before order for read and write operations to shared data items [30]. This relation can order read and write operations across a set of related data items. In particular, if o_1 and o_2 are two

writes to data item x which assign values v_1 and v_2 to x respectively, v_1 is said to be causally overwritten by v_2, if o_1 causally precedes o_2. A secure server that supports CC ensures that no read operation returns a causally overwritten value.

The first consistency model, MRC meets the needs of applications that manipulate personal and non-shared data belonging to a single user and applications in which there is a single writer and multiple readers. The second consistency model, CC, meets the needs of a class of applications that involve asynchronous interaction between multiple readers and writers. Although these models are useful to many applications and efficient to implement, clearly these protocols may not be able to meet the consistency needs of all applications. In particular, some applications may require strong consistency. In this case, existing protocols can be used. For example, the replicated state machine approach based protocol in [7] can be used to ensure that all client operations appear to execute in a total order. MRC and CC do not address how quickly values written by a certain client become available to others. Although models that address timeliness do exist, their implementation in an asynchronous distributed system is infeasible. We do assume that MRC and CC will return newer values eventually when clients continue to read the data.

5.2. Read and Write Protocols for Consistent Access

We present here an extended version of the read and write protocols presented in the earlier section to meet consistency requirements. A similar set of protocols for an environment where only replication is used was presented in [22]. We first discuss some notation and assumptions before we describe the protocols.

Each data item x_i has a unique identifier in the system denoted by $uid(x_i)$. To guarantee consistency, we assume logical clocks at the client sites that share data (e.g, Lamport clocks). A timestamp ts which is derived by concatenating a clockvalue with the unique identifier of the writer serves as a unique identifier of a write. Although the secure store may contain a large number of data items, we assume that each data item belongs to a relatively small group of related data items. For example, documents pertaining to a certain topic may define the group of related data items. We assume that consistency is only required across a group of related data items and not across data items belonging to different groups. X denotes a set of related data items. A set of identifiers and timestamps $((uid(x_1), ts_1), ..., (uid(x_m), ts_m))$, called the *context*, is maintained by each client locally if the client accesses data items in group X. We denote the *context* by $\mathcal{X}$. A server maintains meta-data about each data item for which it stores a share. The meta-data associated with a share of a value includes the unique identifier of the writer, timestamp associated with the write, *context* of the writer at the time data share was written, the associated verification string and other relevant information. We denote the *context* of the writer stored as part of meta-data by $\mathcal{X}_{writer}$. We assume that for a given set of data items, either MRC or CC consistency is specified at the time these data items are first created. Thus, the same set of data items cannot be accessed with MRC consistency at one time and CC consistency at another time.

Context captures a client's interaction with the store in the past. A client uses its *context* information to determine what values are acceptable with respect to the consistency level associated with a given data item. A client, before it starts interaction with the store, initializes its *context* for a given set of data items. Initially *context* consists of null timestamps for all the data items in the set. The *context* is continually updated as the client interacts with the store. At the end of the session, the client saves its *context* so that it can be retrieved and used in a later session. We do not discuss how this *context* is stored. Context can be stored in the same set of servers used to store data using a quorum protocol or can be saved in the client's local stable storage. For the rest of this section, we assume that a client always has a valid *context* for a given data set before it starts its interaction with the store for any data item in that set.

Figure 5 shows the extended version of the read and write protocols with consistency guarantees. These protocols differ from the earlier described set of protocols in that *context* information accompanies all reads and writes. Writes of data shares are accompanied by the *context* information of the writer which is stored along with the share at the servers. Data shares retrieved in a read request are also accompanied by the *context* of the writer that is stored along with the share. Upon a successful read, the local *context* of the reading client is updated with the $context_{writer}$ retrieved along with the shares. At any point, *context* stored locally at the client site contains the oldest possible timestamps of the data items that are acceptable for future reads. Thus, a client will never accept a value for a data item that has a timestamp older than the timestamp stored in its local *context*.

The only difference between MRC and CC consistency is in the way the local *context* is updated with the $context_{writer}$ read along with the data shares. For MRC, only the timestamp of the particular data item being read is updated while for CC timestamps associated with all related data items in the local *context* are updated.

For MRC, a client's *context* always captures the timestamp of its latest write or read for all the data items in the corresponding data set. Thus, clients are always guaranteed

Write(x_j, v) by client C_i, $x_j \in X_j$

1. Let timestamp ts = current clock value concatenated with uid(client). Update $\mathcal{X}_j$ with ts for x_j.
2. Fragment value v into c shares v_1,v_2,..v_c using a (b, c) secret sharing scheme.
3. Compute one-way function of each of the shares, $h(v_i)$.
4. Form the verification string VS and compute signature $VS = h(v_1)|h(v_2)...|h(v_c)$. (concatenation). $sig = \{uid(x_j), ts, v\}_{K_{c_i}^{-1}}$, where $\{\}_{K_{c_i}^{-1}}$ denotes signature using private key of the client.
5. Choose a row k.
   for (m = 1 to c){
   send {"write",uid(x_j),ts,$\mathcal{X}_j$,v_m,VS,sig}
   to server S_{km}.
   }
6. Repeat 5 for a different row until $c - \lfloor b/l \rfloor \geq b + 1$ where l is the number of rows contacted.

Read(x_j) by client C_i, $x_j \in X_j$

1. Let t_j be the time associated with x_j in $\mathcal{X}_j$.
2. Choose a row k.
   for m = 1 to $2b + 1$ {
   send {"read",$uid(x_j)$,t_j} to S_{km}.
   }
3. Receive {t_r,VS,$\mathcal{X}_{writer}$,v_m} from each server.
5. A triplet {t_r,VS,$\mathcal{X}_{writer}$} is said to be "good" if it appears in at least $b + 1$ replies.
   Let t_r be the highest
   timestamp that appears in a good triplet.
6. If there is no good triplet or if $t_r < t_j$, repeat from 2 for a different k.
7. Pick shares corresponding to t_r. Pick $b + 1$ shares among these that are successfully verified by the verification string. Reconstruct the data value.
8. Check if signature is valid. If valid, return the reconstructed data value. If signature is not valid, repeat from 1 for a different k.
9. If MRC consistency is required then
   Update t_j in $\mathcal{X}_j$ to t_r when $t_r > t_j$;
   If CC consistency is required then
   Update each timestamp in $\mathcal{X}_j$ to max of value in $\mathcal{X}_j$ and the corresponding value in $\mathcal{X}_{writer}$;

Figure 5. Write and Read protocols with consistency guarantees

to read a value which is not older than the value already read in the past for that data item.

For CC, during a write, $context$ of the writing client captures the timestamps of all the reads and writes that causally precede that write. This $context$ is stored as $context_{writer}$ along with the data shares. Thus, when a client reads and accepts a data value, it updates its local $context$ with timestamps of all reads and writes that causally preceded the write operation that wrote the value. Hence, in future, the reading client will never accept a value for a data item that is older than a value that causally preceded any of its reads and writes.

Since $context$ itself is accepted as valid only if $b + 1$ servers return the same context, a malicious server cannot make a client accept a spurious $context$ that was not written by any client. While $context$ helps clients avoid reading older values for data items, we assume that newly written data values will eventually be accessible to clients either because of background dissemination or because of retries by clients upon failed reads. In the steady state when no writes are taking place, a read by a client with a valid $context$ is bound to succeed as long as the earlier writes were completed successfully by non-faulty clients.

For MRC, it is not necessary to send the whole $context$ of the data set in a write, just the timestamp of that particular data item alone can be sent. Reading clients would update their local $context$ with the timestamp of the value. To ensure that clients can read values for a data item even when a write is going on concurrently, servers can employ versioning mechanisms to store shares and associated meta-data rather than overwrite shares. Goodson et. al. used a similar technique to deal with client crashes in [33]. However, in this case, servers should return a list of shares and meta-data for a read request.

6. Analysis

In this section, we do an analysis of the secure store based on a probabilistic model and show how the choice of a threshold value and other parameters affect the security and performance of the system. During any continuous time interval of length T_v units, we assume that any server can be compromised with a probability p. Thus, expected number of compromised servers during a time interval of length T_v would be np. However, a lower or higher value can be chosen for b to tolerate certain number of failures depending on whether better performance or stronger security is desired. We assume that the probability of compromising one node is independent of the other. Thus, in the analysis, we do not consider the case of related or similar attacks on nodes operating on same OS or run time code. For the purpose of analysis we also assume that the system is in a steady state, void of concurrent reads and writes. Thus, we assume reads and writes do not fail because of consistency requirements. The analysis we provide here is a simplified one and its goal is to provide us with insights into how some parameters affect security and performance levels offered by the system. In particular, it gives us a direction as to what threshold value and the degree of replication should be chosen, given the

desired levels of various security and performance metrics.

We consider the following security metrics.

- **Availability** : Availability is defined as the probability that a legitimate client can read a data item that has been written successfully.
- **Confidentiality** : Confidentiality is defined as complement of the probability that an adversary can read a data item that has been written successfully.
- **Integrity** : Integrity is defined as complement of the probability that a reading client could be returned corrupted data content without corruption being detected.

We assume that the servers are organized in c columns and r rows with $rc = n$, where n is the total number of servers. Furthermore, we assume that a (b, c) scheme is used for secret sharing. With these assumptions, the security metrics can be evaluated as follows:

Availability(α):

$\alpha(b, c, n)$ = probability of finding at least $b + 1$ non faulty servers, one each from a different column.

$$\alpha = \Sigma_{i=b+1}^{c} \binom{c}{i} (1 - p^r)^i * (p^r)^{(c-i)}$$

Confidentiality(κ):

$\kappa(b, c, n)$ = 1 - probability of finding at least $b + 1$ malicious servers, one each from a different column.

$$\kappa = 1 \; - \; \Sigma_{i=b+1}^{c} \binom{c}{i} (1 - q^r)^i * (q^r)^{(c-i)}, \; q = 1 - p$$

For our system, integrity is same as confidentiality since any compromised node can both disclose data shares and corrupt them. If use of signature is considered, integrity becomes the probability that a signature can be forged. In rest of the section, we discuss only confidentiality and availability as the primary security metrics.

In addition to these security metrics, for our system, we also define the following performance metrics.

- **Read cost** : Read cost is defined as the expected number of servers a client needs to contact to read a data item successfully. A data item is read successfully upon collecting $b + 1$ distinct shares for the data item.
- **Write cost** : Write cost is defined as the number of servers a client needs to contact to write a data item at a confidence level *h*. By confidence level, we mean the probability that a write has been successfully completed, that is, at least one non-faulty server from each of $b + 1$ or more columns has registered the write.

Both read and write costs are defined in terms of number of servers contacted. We expect the communication cost to be the dominant factor compared to the computation cost and hence discuss only communication costs in this section. This may not be true when we consider large data items. Secret sharing and data reconstruction are costly for such items. Furthermore, computation cost also increases with threshold value. Number of messages sent/received during a read or write operation is twice the number of servers contacted to complete the operation.

Read cost is calculated based on the following protocol: A client contacts $2b + 1$ servers, each from a different column. If it has successfully collected $b + 1$ shares, read returns. Otherwise, client contacts additional servers as necessary. If p_r is the probability that a client would find $b + 1$ non-faulty servers among the $2b + 1$ it contacts, then expected number of servers a client needs to contact is approximately $(2b + 1)/p_r$.

When a client does a write, it writes to some number of rows so that the probability of finding at least one non-faulty server from each of $b+1$ or more columns is greater than or equal to h where h is the confidence level. If $p_w(k)$ is the probability of a write being successful when a client writes to k rows, then write cost is $\mathbf{wc} * c$ where $\mathbf{wc}$ is such that $p_w(\mathbf{wc} - 1) < h$ and $p_w(\mathbf{wc}) \geq h$.

Given the probability of a node being compromised and the total number of servers, there are two parameters that determine the values of the security and performance metrics. These are the threshold level b and the degree of replication which is the number of rows or alternatively the number of columns c.

We calculated the four metrics by varying these two parameters for a system of 45 servers with $p = 0.15$.

Plots 6(a), 6(c) and 6(e) in figure 6 show availability and confidentiality. Both availability and confidentiality are plotted in logarithmic scale. For a value x plotted in the graph, the corresponding probability (availability or confidentiality) is $1 - 10^{-x}$. Plots 6(b), 6(d) and 6(f) show read and write costs. For plots 6(a) and 6(b), number of columns was varied for fixed values of threshold. For graphs 6(c) and 6(d), threshold value was varied for fixed values for number of columns. For graphs 6(e) and 6(f), threshold value was varied and number of columns was set to $2b + 1$ accordingly.

We observe the following dependencies :

- For a given threshold level, increasing the number of columns (and hence decreasing the number of rows) increases availability and decreases confidentiality. Write cost increases linearly with number of columns but read cost remains almost a constant.
- For a given number of columns (and rows), increasing the threshold value decreases availability and increases confidentiality. Read cost increases linearly with threshold. Write cost remains almost a constant for low threshold levels. As threshold value ap-

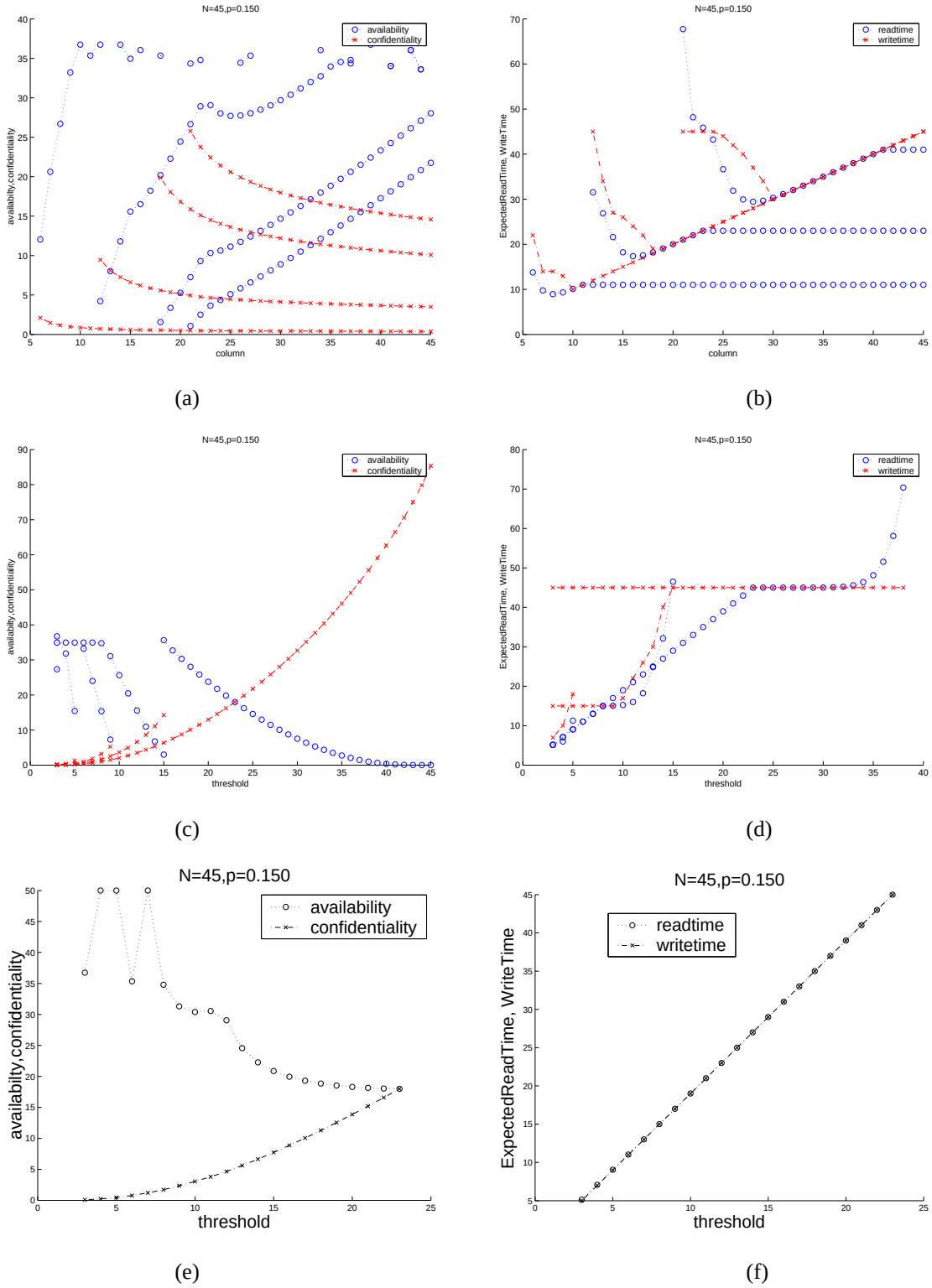

Figure 6. (a) α,κ as functions of number of columns for various threshold values b = 1,11,17,20, (b) access costs as functions of c for various threshold values b = 1,11,20, (c) α,κ as functions of threshold value for various c values, c = 3,5,9,15,45, (d) access costs as functions of threshold value for various c values c = 5,15,45, (e) α,κ as functions of threshold value for $c = 2b + 1$, (f) access costs as functions of threshold value for $c = 2b + 1$. A probability value of $1 - 10^{-x}$ for α or κ is plotted as x.

proaches the number of columns, write cost starts increasing.

- By choosing c value optimally for every b value, increasing b increases access costs and confidentiality and decreases availability.

We can see from the plots in figure 6 that a hybrid scheme provides more flexible design options for the secure store. Given p, the likelihood of a node being compromised, the choice of the degree of replication and secret sharing depends on the security and performance metrics that need to be optimized. Clearly, when access cost or availability is of paramount importance, pure replication ($r = n, c = 1, b = 0$) is the best option. On the other hand, when confidentiality is the critical metric, pure secret sharing ($r = 1, c = n, b = \lfloor (c-1)/2 \rfloor$) is the best option. There is a wide range of confidentiality, availability and access costs where the desired levels are achieved when the server nodes are arranged in a certain number of rows and columns. Thus, both replication and secret sharing are essential when certain bounds are placed on the security and performance metrics. For example, with a confidentiality requirement of $\kappa \geq 1 - 10^{-3.5}$, when access cost should not exceed 22 servers, the optimal choice would be $b = 10, c = 21$.

For a given assumption on p, the security level (confidentiality and integrity) can be increased by sacrificing availability and low access costs. On the other hand, decreasing the threshold level results in improved performance and better availability but exposes the stored data to a higher risk of being compromised. Thus, our analysis demonstrates that our hybrid scheme offers greater flexibility in meeting performance and security goals of a secure store.

Apart from the flexible tradeoff our system offers between security and performance, one additional benefit in our system is tolerance to related attacks. When nodes vulnerable to related attacks are placed in the same column, stealing information is not any easier for an adversary.

7. Conclusion

In this paper, we have presented a design for a distributed store that integrates replication and secret sharing to meet both performance and security needs while tolerating Byzantine faults. By our simple analysis, we have shown that for a probabilistic model of Byzantine failures, our design provides greater flexibility in meeting security and performance needs compared to either a pure replication or a pure secret sharing scheme. Although we used specific secret sharing and share renewal schemes for the discussion in this paper, the design of our system does not depend on any specific choice for these schemes. We could use any secret sharing or share renewal scheme that satisfies certain requirements, as appropriate for our protocols.

There are several directions in which our research can be pursued further. First, we are currently experimenting with and exploring other possible secret sharing schemes and one-way functions for our purpose. Currently, our system provides the same level of security and performance for all objects in the store. This could be further extended to provide flexible security guarantees, customizable on a per-object basis. Dynamic inclusion and exclusion of servers and secure authorization service are some interesting problems to explore. In the future, we also plan to prototype a secure store and evaluate it experimentally.

References

[1] The Aware Home Research Initiative, http://www.cc.gatech.edu/fce/ahri/, 2000.

[2] Malkhi D., Reiter M., “Secure and Scalable Replication in Phalanx”, *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[3] Shamir A., “How to Share a Secret”, *Communications of the ACM*, 22, 1979, pp.612–613.

[4] Lamport L., Shostak R., Pease M., “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[5] Herzberg A., Jarecki S., Krawczyk H., and Yung M., ”Proactive secret sharing,” *Advances in Cryptology, Crypto ’95*, Lecture Notes in Computer Science, 963, D. Coppersmith, Ed., 1995, Springer-Verlag.

[6] Schneider F., “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”, *ACM Computing Surveys*, Vol. 22, No. 4, December 1990.

[7] Castro M., Liskov B., “Practical Byzantine Fault Tolerance”, *Proc. 3rd Symposium on Operating Systems Design and Implementation, New Orleans*, Feb. 1999.

[8] Scalable and Survivable Data Replication : The Fleet Project, http://www.bell-labs.com/user/reiter/fleet/.

[9] Alvisi L., Malkhi D., Pierce E., Reiter M., Wright R., “Dynamic Byzantine Quorum Systems”, *Proc. International Conference on Dependable Systems and Networks*, June 2000.

[10] Blakley G., ”Safeguarding Cryptographic Keys”, *Proc. Nat’l Computer Conf.*, American Federation of Information Processing Societies, Montvale, N.J., 1979, pp. 313-317.

[11] Rabin M., “Efficient dispersal of information for security, load balancing and fault tolerance, *Journal of the ACM*,36(2):335-348, 1989.

[12] Tompa M.and Woll H., ”How to Share a Secret with Cheaters,”, *Journal of Cryptology*, Feb. 1988.

[13] A. De Santis and B. Masucci, ”Multiple Ramp Schemes,” *IEEE Trans. Information Theory*, July 1999, pp. 1720-1728.

[14] Feldman P., “A practical scheme for non-interactive verifiable secret sharing”, in *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science*, IEEE Press, 1987, 427-437.

[15] Pederson T.P., “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”, *Proc. Crypto’ 91*, LNCS 576, pp. 129-140.

[16] Wylie J.J,, Bigrigg M.W., Strunk J.D, Ganger G.R., Kiliccote H., and Khosla P.K., “Survivable information storage systems”,*IEEE Computer*, 33(8):61-68, August 2000.

[17] Demers A., Greene D., Hauser C., Irish W., Larson J., Shenker S., Sturgis H., Swinehart D, and Terry D., “Epidemic algorithms for replicated database maintenance”, *Proc. 6th Symposium on Principles of Distributed Computing*, pp. 1-12,1987.

[18] Malkhi D., Mansour Y., and Reiter M., “On diffusing updates in a Byzantine environment”, *Proc. 18th IEEE Symposium on Reliable Distributed Systems*, Oct. 1999.

[19] Malkhi D., Pavlov E., Sella Y., “Optimal Unconditional Information Diffusion”, *Proceedings of the 15th International Symposium on Distributed Computing*, pages 63-77, Lisbon, Portugal, 2001.

[20] Minsky Y., Schneider F., “Tolerating Malicious Gossip”, to appear in *Distributed Computing*, Technical Report, Cornell Computer Science TR2001-1853.

[21] Fray JM., Deswarte Y. and Powell D., “Intrusion-tolerance using fine-grain fragmentation-scattering”, *Proc. 1986 IEEE Symposium on Security and Privacy*, Oakland (CA), April 1986.

[22] Lakshmanan S., Ahamad M., H.Venkateswaran, “A secure and highly available distributed store for meeting diverse data storage needs”, *Proc. 2001 International Conference on Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[23] Maurice P. Herlihy and J.D.Tygar, “How to make replicated data secure”, *Crypto ’87*.

[24] Hugo Krawczyk, “Secret sharing made short”, in *Advances in Cryptology – CRYPTO ’93*, D. R. Stinson, ed., Lecture Notes in Computer Science 773 (1994), 136-146.

[25] Hugo Krawczyk, “Distributed fingerprints and secure information dispersal”, *12th ACM Symposium on Principles on Distributed Computing*, Ithaca, NY, 1993.

[26] Moni Naor and Avishai Wool, “Access Control and Signatures via Quorum Secret Sharing”, *IEEE Trans. Parallel and Distributed Sys* 9(9), 1998.

[27] Lamport L., “On Interprocess Communication”, *Distributed Computing*, 1:77–101, 1986.

[28] Lamport L., “How to make a multiprocessor computer that correctly executes multiprocessor programs”, *IEEE Transactions on Computers*, 28(9):690-691, 1979.

[29] Herlihy M. and Wing J., “Linearizability: A Correctness Condition for Concurrent Objects”, *ACM Transactions on Programming Languages*,12(3), July 1992.

[30] Ahamad M., Neiger G., Burns J., Hutto P., Kohli P., “Causal Memory: Definitions, Implementations and Programming”, *Distributed Computing journal*, Springer-Verlag, Aug. 1995.

[31] Terry D., Demers A., Peterson K., Spreitzer J., Theimer M., Welch B., “Session Guarantees for Weakly Consistent Replicated Data”, *Proc. International Conference on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[32] Yu H. and Vahdat A., “Design and Evaluation of a Continuous Consistency Model for Replicated Services”, *Proc. Operating Systems Design and Implementation*, October 2000.

[33] Garth Goodson, Jay Wylie, Greg Ganger and Mike Reiter, “Decentralized Storage Consistency via Versioning Servers”, *Carnegie Mellon University Technical Report CMU-CS-02-180*, September 2002.