# An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments*

Arun Subbiah and Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332 USA
{arun,dblough}@ece.gatech.edu

Technical Report GIT-CERCS-05-13
April 2005

**Abstract**

We describe a novel approach for building a secure and fault tolerant data storage service in collaborative work environments. In such environments, sensitive data must be accessible only to a select group of people, whose membership may change over time. Key management issues are a recognized problem in such environments. We eliminate this problem for confidential and secure data storage by using perfect secret sharing techniques for storing data. Perfect secret sharing schemes have found little use in managing generic data because of the high computation overheads incurred by existing schemes. Our proposed approach uses a novel combination of XOR secret sharing and replication mechanisms, which drastically reduce the computation overheads and achieve speeds comparable to standard encryption schemes. The combination of secret sharing and replication manifests itself as an architectural framework, which has the attractive property that its dimension can be varied to tradeoff amongst different performance metrics. We evaluate the properties and performance of the proposed framework to show that the combination of perfect secret sharing and replication can be used to build efficient fault-tolerant and secure distributed data storage systems for collaborative work environments.

**Keywords:** Distributed data storage, secret sharing, replication, confidentiality, Byzantine fault tolerance, collaborative environments

## 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 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 personal information such as cryptographic keys and passwords, to generic data that must be available to only a select group of people. This paper describes a novel fault-tolerant and secure distributed storage system designed for use in collaborative work environments, where stored data is shared by a group of people whose membership may change over time.

The traditional approach for storing data securely and reliably is to encrypt the data for confidentiality, and store the encrypted data using replication-based techniques for fault tolerance. This approach has the benefits of

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

being computationally and storage efficient. However, when data must be stored for extended periods of time, it can be expected that there will be changes in the list of users authorized to read or write the encrypted data. Changes in the access list will require re-encrypting the stored data with a new cryptographic key, which may be cumbersome. For fine grained access list management, each file or document stored at the data storage service would require a unique key. The number of keys could then become large, and the keys would then have to be stored at the data storage service itself for easy access by authorized users. Obviously, these keys must be stored at the storage service in a secure and fault tolerant manner without using additional keys.

We solve these problems by using a combination of replication-based schemes and perfect secret sharing techniques for storing data. Perfect secret sharing schemes encode data into *shares* such that only certain valid combinations of shares can be used to reconstruct the encoded data, while invalid combinations of shares give no information on the encoded data. By storing these shares at different servers, the encoded data is kept confidential as long as not enough servers are compromised. Confidentiality is achieved without encryption, thus avoiding the need for the storage and management of cryptographic keys. Perfect secret sharing schemes have the additional property that the shares can be changed, or "renewed", distributively such that the encoded data still remains the same. This process of share renewal, when performed often, can provide strong data confidentiality.

On the other hand, unlike private-key encryption schemes, most perfect secret sharing schemes are computationally expensive. Verifiable secret sharing schemes are typically used with perfect secret sharing schemes to detect incorrect shares that may be returned by faulty or compromised servers. Such techniques further increase the computation time during the encoding and decoding of data. We solve these problems by 1) using XOR secret sharing for fast computations, and 2) using replication-based schemes to detect incorrect shares that may be returned by faulty or malicious servers. This combination of secret sharing and replication manifests itself as an architectural framework, where servers are arranged in the form of a rectangle or a grid. The proposed architectural framework, which we call *GridSharing*, has the useful property that its dimensions can be varied to trade off several performance metrics.

The proposed approach is also useful for storing sensitive archival data. Encryption techniques may not be suitable for such purposes, as in the long run, encryption algorithms can be broken, or increased computing resources would require increasing the sizes of the keys. In our proposed framework, long-term confidentiality can be provided by performing share renewal often. Another drawback of encryption is the assumption that the data to be encrypted is random in nature, while that is often not the case. Encrypted data could be subject to cryptanalysis, and it may be possible for an adversary to obtain some information on the encrypted data. In our proposed framework, we assume that not more than a threshold servers are compromised (between two consecutive share renewals), and use perfect secret sharing schemes, such as the XOR secret sharing scheme, which are information theoretically secure. Thus, even if an adversary obtained shares from a threshold servers, he still cannot obtain any information on the encoded data.

Our contributions are as follows: We describe a novel approach for building a secure and fault tolerant data storage service in collaborative work environments. Key management issues are a well-known problem in such environments, where data may be shared by a group of people whose membership may change over time. We eliminate this problem for confidential and secure data storage by using perfect secret sharing techniques for storing data. Perfect secret sharing schemes have found little use in managing generic data because of the high computation overheads such schemes incur especially when supplemented with mechanisms to achieve Byzantine fault tolerance. Our proposed approach uses a novel combination of XOR secret sharing and replication mechanisms, which drastically reduce the computation overheads and achieve speeds comparable to standard encryption schemes. The combination of secret sharing and replication manifests itself as an architectural framework, whose dimension can be varied to tradeoff amongst different performance metrics. We evaluate the properties and performance of the proposed framework to show that the combination of perfect secret sharing and replication can be used to build efficient fault-tolerant and secure distributed data storage systems for collaborative work environments.

## 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.

In this paper, we use secret sharing schemes to provide data confidentiality. Most works use imperfect secret sharing schemes, such as 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 are thus not information-theoretically secure, but allow savings in storage space. Given enough time, an adversary may compromise enough servers to learn the encoded data. Thus, to provide long-term confidentiality, the secret sharing scheme should allow share renewal, where the shares are changed in a distributed fashion such that the encoded secret is not recovered in the process and is unchanged. To our knowledge, no share renewal scheme for imperfect secret sharing has been developed to date. Perfect secret sharing schemes, on the other hand, allow share renewal. Perfect secret sharing schemes are information-theoretic secure, meaning the leakage of an insufficient number of shares to an adversary does not reveal any information on the encoded data.

Several works have combined replication and perfect secret sharing [2, 15, 16]. [2] presents a scheme where data is encrypted using a key, and both are stored at the storage servers. The data is stored in replicated form in a quorum, while the key is stored using secret sharing. [15] 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 stored are access rights, and quorum properties are used to grant or revoke access rights. Thus, these two works consider the use of perfect secret sharing for some special types of data and not for generic data. Performance during reads and writes is not addressed. [16] considers perfect secret sharing schemes for generic data. They use a verifiable secret sharing scheme along with replication for high availability. Their work primarly addresses the overheads associated with data dissemination, but does not address the performance issues with using verifiable secret sharing schemes and perfect secret sharing schemes. [17] uses perfect secret sharing along with verifiable secret sharing for storing archival data. Extensive performance measurements of such schemes are given, but the problem of high computation overheads is not addressed.

CODEX [18] is a data storage system that avoids key management issues by encrypting secrets using the public key of the data storage service. The private key is secret shared at the data storage servers so that up to some threshold malicious servers can collude and still not be able to recover the service's private key to decrypt the data. They report their computation overheads to be in the hundreds of milliseconds for 128 byte data secrets, which is over a hundred times slower than our measurements of the Rijndael encryption algorithm. Though we have not implemented their approach and compared their performance against ours, we note that due to their use of expensive cryptographic operations, their computation latencies are expected to be much higher than our approach.

XOR secret sharing has been considered in [19]. They show how different capabilities such as share renewal and share recovery can be implemented with XOR secret sharing. For this, existence of a trusted device, called the *Accumulator*, is assumed. They further assume that during secret sharing and secret recovery, no server returns erroneous responses. Performance benefits associated with the use of XOR secret sharing are not discussed.

## 3 Background

### 3.1 Secret Sharing Schemes

Secret sharing schemes are techniques where a *secret* is encoded into several fragments, called *shares*, such that certain combinations of shares can together reveal the encoded secret. In *perfect* secret sharing schemes, invalid or unauthorized combinations of shares give no information on the encoded secret. Thus, perfect secret sharing

schemes are information-theoretic secure. Perfect secret sharing schemes also allow share renewal, which is the process of distributively changing the shares such that the encoded secret is the same. Share renewal must be done often to provide strong data confidentiality.

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

In the next subsection, we describe Ito, Saito, and Nishizeki's share assignment scheme [20], which realizes any access structure using a $(q, q)$-threshold secret sharing scheme.

### 3.2 Ito, Saito, and Nishizeki's Share Assignment Scheme

We describe Ito, Saito, and Nishizeki's share assignment scheme [20] for a threshold access structure. Consider a set of $r$ participants $\{P_1, P_2, ..., P_{\mathrm{r}}\}$ such that any $(m+1)$ participants can pool their shares to recover the encoded secret. For a secret sharing scheme realizing this access structure, first list the set $B$ consisting of all possible combinations of $m$ participants. Thus, $B = \{B_1, B_2, ..., B_{\mathrm{q}}\}$, where $q = \binom{r}{m}$.

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

For example, consider a set of four participants such that at least three participants must pool their shares to find the encoded secret. Then $r = 4$, $m = 2$, and the set $B = \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_3), (P_2, P_4), (P_3, P_4)\}$. Next, generate $6$ shares of the secret such that all six of them are needed to decode the secret. Denote the six shares by $\{s_1, s_2, s_3, s_4, s_5, s_6\}$.

From the share assignment function $g$,

Participant $P_1$ gets shares $(s_4, s_5, s_6)$,
Participant $P_2$ gets shares $(s_2, s_3, s_6)$,
Participant $P_3$ gets shares $(s_1, s_3, s_5)$,
Participant $P_4$ gets shares $(s_1, s_2, s_4)$.

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

## 4 Computation Overhead of Perfect Secret Sharing Schemes

In this section, we show the high computation overhead of some well known secret sharing schemes, which is the main reason why such schemes are not widely used in secure and fault tolerant distributed data storage systems. We contrast the computation overheads with that of the Rijndael (AES) symmetric-key encryption algorithm to illustrate this point. We then show that XOR secret sharing combined with replication-and-voting mechanisms has a computational overhead similar to that of Rijndael. All performance measurements reported in this paper were done on an Intel Pentium4 3GHz processor with $256$ MB RAM running Linux $2.6.9$. The MIRACL [21] library was used to implement the cryptographic algorithms.

Shamir's scheme [1] is an example of a $(k, q)$-threshold perfect secret sharing scheme, where $k \leq q$. Table 1 lists the time taken to compute shares (sharing), and the time taken to compute the secret given enough shares

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

**Table 1: Computation time during secret sharing and secret recovery for an 8 KB block of data using Shamir's secret sharing scheme**

(recovery), for an $8$ KB block of data using Shamir's scheme, for a selection of $(k, q)$ values. Secret sharing and recovery are done during writes and reads, respectively, and their overheads are therefore important. For Shamir's scheme, since the computations are done modulo a prime $p$, the size of this modulus is also a factor in the throughput measurements.

Shamir's scheme alone cannot be used to detect incorrect shares returned by malicious servers during reads. One method of detecting incorrect shares returned by malicious servers is distributed fingerprints [22], where the hash of the shares are stored as a hash vector at all servers. However, share renewal cannot be used with this approach because it is not possible to update the hash vector during a distributed renewal of the shares. Another technique to safeguard against malicious servers is verifiable secret sharing. In such schemes, some common data for all the shares is computed and stored at all the servers. During reads, the correct common data is first determined, and then each share is checked against this common data to detect incorrect shares. With verifiable secret sharing schemes, it is possible to perform a distributed share renewal of the shares and the additional data required for verifiability. A widely used method for verifiable secret sharing is Feldman's scheme [23]. Table 2 gives the computation times during secret sharing and secret recovery of an $8$ KB block of data when Feldman's scheme is used with Shamir's scheme.

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

**Table 2: Computation times for verifiable secret sharing and verifiable secret recovery for an 8 KB block of data using a combination of Shamir's and Feldman's secret sharing schemes. Computations in Feldman's scheme were implemented modulo a prime of length 1025 bits.**

For comparison purposes, the throughputs of the AES Rijndael symmetric-key encryption algorithm are given in Table 3. The implementation of Rijndael in the MIRACL library [21] was used for the measurements.

| Key length | Encryption | Decryption |
|---|---|---|
| 16 bytes | 205 $\mu$s | 205 $\mu$s |
| 24 bytes | 230 $\mu$s | 241 $\mu$s |
| 32 bytes | 282 $\mu$s | 271 $\mu$s |

**Table 3: Time taken to encrypt and decrypt an 8 KB block of data using the Rijndael (AES) encryption scheme in CBC mode.**

From Tables 1–3, it is clear that the computation times of Shamir's scheme and Feldman's scheme are far higher than those of symmetric-key encryption and, in fact, this performance is well below what is acceptable for modern data storage systems. The secret recovery computation time of verifiable secret sharing schemes are at least $3000$ times slower than the Rijndael decryption times. The above analyses also indicate, in part, why perfect secret sharing techniques have not been adopted for generic data to date. However, as mentioned previously, perfect secret sharing has several benefits as compared to encryption-based techniques: it provides information-theoretic secrecy, the shares can be renewed for strong data confidentiality, and there are no cryptographic keys to be secured and managed. To reduce the computation overheads incurred during perfect secret sharing, we employ the following two mechanisms:

**Mechanism 1: Use a (q, q) perfect secret sharing scheme:** When $k = q$, i.e., all the shares are needed to recover the secret, then a simple bit-wise XOR secret sharing can be used. If each data bit is thought of as a separate secret, then each share is a single bit and XOR of the $q$ shares (or $q$ bits) gives the encoded secret bit. In practice, XOR secret sharing can be implemented with word-wide operations for efficiency. Table 4 lists the computation times during secret sharing and secret recovery for a selection of $(q, q)$ values for XOR secret sharing. Note that XOR secret sharing is also a perfect secret sharing scheme. The only constraint compared to the general $(k, q)$-threshold scheme with $k < q$ is that all $q$ shares must be recovered to reconstruct the secret. Compared with the computation

| $(q, q)$ | Secret sharing | Secret recovery |
|---|---|---|
| $(5, 5)$ | $333$ $\mu$s | $35$ $\mu$s |
| $(10, 10)$ | $732$ $\mu$s | $60$ $\mu$s |
| $(20, 20)$ | $1.494$ ms | $140$ $\mu$s |

**Table 4: Computation times for secret sharing and secret recovery of an 8 KB block using the XOR secret sharing scheme**

times using Shamir's scheme (Table 1), the computation times using XOR secret sharing are much lower.

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

| $b$ | Computation Time |
|---|---|
| 1 | $13.75$ $\mu$s |
| 2 | $25$ $\mu$s |
| 3 | $40$ $\mu$s |
| 4 | $50$ $\mu$s |
| 5 | $65$ $\mu$s |

**Table 5: Computation times for voting out of 2b+1 responses to determine a share of size 8 KB. b is the maximum number of malicious servers that can return incorrect values for the requested share. Measurements reflect the best case where there are no incorrect responses.**

Table 5 gives the computation times for determining each share from $(2b+1)$ responses, where $b$ is the number of possibly malicious servers. Note that the numbers are given for each share. Hence, the computation time during

secret recovery must now include the product of the time taken to determine each share from $(2b + 1)$ responses and the number of shares. The secret sharing computation time will remain unchanged as no additional shares are generated. The secret sharing and recovery computation times for XOR secret sharing along with voting for $b = 3$ are shown in Table 6. Compared with the computation times of verifiable secret sharing schemes (Table 2), the

| $(q, q)$ | Secret sharing | Secret recovery |
|---|---|---|
| $(5, 5)$ | 333 $\mu$s | 235 $\mu$s |
| $(10, 10)$ | 732 $\mu$s | 460 $\mu$s |
| $(20, 20)$ | 1.494 ms | 940 $\mu$s |

**Table 6: Computation times for secret sharing and secret recovery of an 8 KB block using the XOR secret sharing scheme along with voting to determine incorrect shares that may be returned by up to b = 3 malicious servers during reads.**

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

Summarizing, perfect secret sharing schemes can be used for fault-tolerant and secure distributed data storage by combining them with verifiable secret sharing techniques. Using the computation latency of the Rijndael encryption algorithm as the benchmark, we have shown that well known verifiable secret sharing techniques such as the combination of Feldman's scheme with Shamir's scheme are too slow to be used for large volumes of data. Verifiable secret sharing techniques can be avoided by using replication-and-voting mechanisms. This, along with the use of XOR secret sharing, drastically lowers the computation times, making them comparable to the Rijndael encryption algorithm execution times. In the rest of the paper, we describe in detail how XOR secret sharing with replication-and-voting mechanisms can be combined, and the benefits of this approach.

# 5 Fault and Adversary Model

Since our data storage service must offer availability, integrity, and confidentiality guarantees for the stored data, we identify the following three types of server faults:

- **Crash:** A server is said to be *crashed* if it stops performing all computations and neither sends nor receives messages on the network.
- **Byzantine:** A Byzantine-faulty server can deviate arbitrarily from its specified protocol. A Byzantine faulty server can also reveal the shares stored locally and its internal state to an adversary.
- **Leakage-only:** A server is said to exhibit a leakage-only fault if it can reveal its shares and state to an adversary, but executes its specified protocol faithfully.

Crash and Byzantine faults are primarily related to process behavior, and hence used widely in fault tolerant distributed algorithms. Works in computer security instead view faults as due to an adversary, and classify the adversary's intent. A *passive* adversary compromises servers only to learn the locally stored data or the servers' states. An *active* adversary, on the other hand, takes full control of a compromised server. An active adversary can learn the compromised server's data and its state, and can also force the compromised server to behave maliciously. The proposed fault model merges these two traditional fault models. Crash faults are primarily for fault tolerance. Byzantine faults take into account compromises due to an active adversary. The leakage-only faults are due to a passive adversary. A faulty server's behavior can belong to only one of the three fault types. In particular, a server cannot simultaneously crash and be leakage-only faulty.

The reason for including a leakage-only fault (or the reason for including passive and active adversary models) is because an adversary may be interested in learning the stored data and avoid generating suspicious activity that might allow the security breach to be detected. Byzantine-fault tolerant algorithms are usually costly, and so it should be desirable to treat this fault separately from the leakage-only fault. For example, in the context of the proposed framework, the secret sharing scheme could be designed such that a substantial number of shares are needed to recover the encoded data, but the Byzantine-fault tolerant read and write protocols could use a small Byzantine fault threshold to reduce overheads.

The proposed fault model is also useful in analyzing works in computer security. Works in computer security almost always do not consider crash faults, thereby overlooking some fault tolerance properties provided naturally by many algorithms. For example, in a $(k, q)$ access structure where $k$ out of $q$ servers must be queried to obtain a threshold shares for secret recovery, up to $(q - k)$ servers can crash, and up to $(k - 1)$ servers can be leakage-only faulty. The crash fault tolerance is usually overlooked when specifying the confidentiality aspects of the access structure. To achieve Byzantine fault tolerance amongst the same $q$ servers, the crash and leakage-only fault tolerances will in general have to be lowered. In storage systems where data is encrypted and then replicated at multiple sites, all the sites can be leakage-only faulty, as the confidentiality of the encrypted data rests on the secure maintenance of the decryption key, which could be stored elsewhere. The proposed fault model can thus be used as a common ground for reasoning about works in fault tolerance and security, and in recognizing the fault tolerance and security properties of works exclusively in either of these two areas.

We use the threshold fault model for each of the three types of faults. We assume that not more than $c$ servers can crash, not more than $b$ servers can be Byzantine-faulty, and not more than $l$ servers can exhibit leakage-only faults.

# 6 Combining Secret Sharing and Replication: The GridSharing Framework

Our approach for a fault tolerant and secure data storage service is to use perfect threshold secret sharing for data confidentiality, and to use replication-based mechanisms to manage each share for crash and Byzantine fault tolerance. This section describes the architectural framework, called *GridSharing*, that combines these two principles.

The *GridSharing* framework consists of $N$ servers, where not more than $c$ servers can crash, not more than $b$ servers can be Byzantine faulty, and not more than $l$ servers can exhibit leakage-only faults. The $N$ servers are arranged in the form of a logical rectangular grid with $r$ rows and $\frac{N}{r}$ columns, where for simplicity it is assumed that $N$ is a multiple of $r$. The arrangement is depicted in Figure 1.

Servers in the same row store replicas of the same shares. Thus, tolerance to crash and Byzantine failures is achieved. Data confidentiality is achieved using secret sharing. The secret sharing is done across rows. Thus, as per the terminology used in Section 3.2, the $r$ rows are the $r$ participants amongst which shares are distributed. Since up to $l$ servers can be leakage-only faulty (reveal their shares to an adversary) and up to $b$ Byzantine-faulty servers can also do the same, shares from up to $(l + b)$ rows can be disclosed to an adversary. From Section 3.2, an $\left(\binom{r}{l+b}, \binom{r}{l+b}\right)$-threshold perfect secret sharing scheme can be used to tolerate $(l + b)$ faulty servers in $r$ rows.

Figure 1 gives an example where $N = 20$ servers are arranged in a rectangular grid with $r = 4$ rows. If it is necessary to tolerate $b = 1$ Byzantine fault and $l = 1$ leakage-only fault, then a $\left(\binom{4}{2}, \binom{4}{2}\right) = (6, 6)$ XOR secret sharing scheme will have to be used. Assume a secret file $S$ is encoded into six file shares $(s_1, s_2, s_3, s_4, s_5, s_6)$ such that $S = s_1 \oplus s_2 \oplus s_3 \oplus s_4 \oplus s_5 \oplus s_6$. That is, each bit in the file $S$ is the XOR of the corresponding bits in the files $s_1, s_2, s_3, s_4, s_5, s_6$. Then according to the share assignment function $g$ given in Section 3.2,

Servers in row 1 gets shares $(s_4, s_5, s_6)$,
Servers in row 2 gets shares $(s_2, s_3, s_6)$,
Servers in row 3 gets shares $(s_1, s_3, s_5)$,

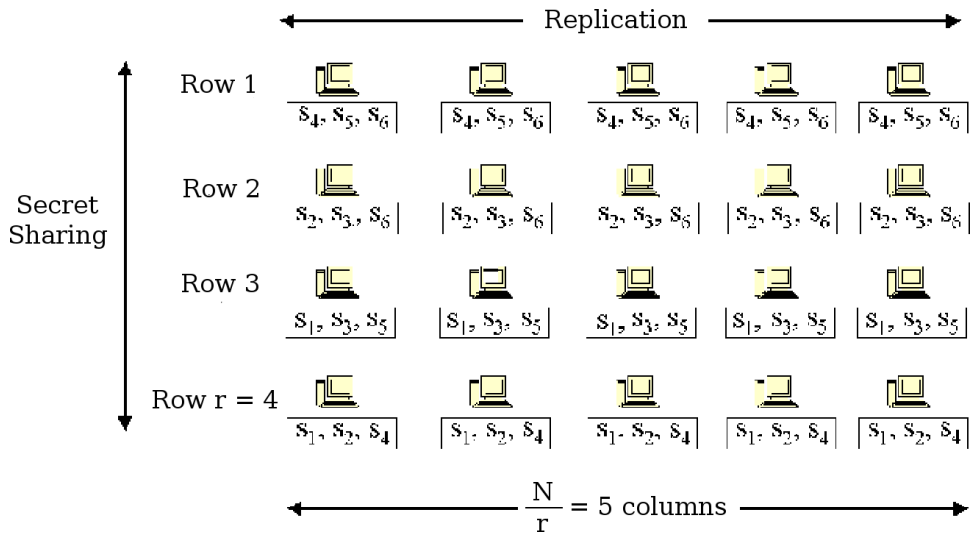


**Figure 1: The *GridSharing* framework: N servers are arranged in a logical grid having r rows. The secret sharing is done across rows, and shares are replicated along rows. The figure shows the setup for N = 20 servers, leakage-only fault threshold l = 1, Byzantine fault threshold b = 1, and crash fault threshold c = 6. Note that each server holds 3 shares.**

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

Note that shares are replicated along rows. The replication is done to achieve Byzantine and crash fault tolerance. When files are read and written, the shares are read and written using replication-based protocols. For the purposes of this analysis, we assume the following simple replication protocol. To write a secret file $S$, the user generates $\binom{r}{l+b}$ files (shares) such that their bitwise-XOR gives the secret file $S$. For a given grid configuration, the share assignment for each server is given by the share assignment function $g$ and the fact that servers in the same row store the same shares. The user writes to each server its assigned shares. Thus, in the example depicted in Figure 1, the user will write to each server in row $1$ the shares $(s_4, s_5, s_6)$, to each server in row $2$ the shares $(s_2, s_3, s_6)$, and so on.

When file $S$ is to be read at a later time, the same user or a different user will need to only contact some set of servers to read all the shares. Consider how share $s_4$ is read in our example. The share $s_4$ is stored at rows $1$ and $4$, and since each row has five servers, the share $s_4$ is stored at ten servers. The user needs to contact only $(2b+1)$ of these servers to determine share $s_4$, since only a maximum of $b$ servers can be Byzantine faulty. The share $s_4$ returned by at least $(b+1)$ servers should have been returned by at least one server that is not Byzantine-faulty, and therefore should be correct. The user must obtain at least $(2b+1)$ responses to determine share $s_4$, but up to $(c+b)$ servers can fail to return any response. Assuming clients connect to the servers over an asynchronous network so that they are unable to detect server failures, each share must be written to at least $((2b+1)+(c+b)) = (3b+c+1)$ servers for reads to be successful in the presence of $b$ Byzantine failures and $c$ crash failures in the system.

Thus, each share must be stored on at least $(3b + c + 1)$ servers. Note that the given description for writes and reads is only an approach for a possible replication-based protocol to manage the shares. We have overlooked the need for the use of timestamps which are common to all the shares. All the shares must be written as part of a single write operation. The approach described is just sufficient to derive a lower bound on the number of servers required to store each share. This lower bound will change based on the assumptions on the system model and the kind of read-write semantics to be realized. The minimum number of servers needed to maintain each share is the only point in the design of the framework that is dependent on the choice of the replication protocol and its

underlying assumptions.

In the proposed framework, each share is assigned to $(r-(l+b))$ rows, and each row has $\frac{N}{r}$ servers. Thus, each share is stored at $(r-(l+b))\frac{N}{r}$ servers, and this must be at least $(3b+c+1)$. Thus,

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

which gives

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

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

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

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

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

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

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

# 7 Performance Analysis of GridSharing

## 7.1 Performance Metrics

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

- **min(N)**: is the minimum number of servers required for a given $l$, $b$, $c$, and $r$. This is given by the smallest $N$ satisfying Inequality 3, with $N$ being a multiple of $r$.
- #**Shares**: The total number of shares generated per data block (or secret). For the proposed framework, #Shares $=\binom{r}{l+b}$.
- **Storage Blowup Per Server**: is defined as the ratio of the amount of storage space taken at each server to the size of the data encoded. For the proposed framework, the storage blowup factor is $\binom{r-1}{l+b}$. Since we use the XOR secret sharing scheme, the size of a share is the same as the size of the secret.
- **Secret Sharing and Secret Recovery Computation Times**: The secret sharing computation time is the time taken to generate (#Shares) shares of an 8 KB block of data. The secret recovery computation time is

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

### 7.2 Effect of Grid Dimension on Performance Metrics

For given security and fault tolerance thresholds $l$, $b$, and $c$, the performance metrics can be traded off against each other by varying the number of rows $r$ in the framework. The secret sharing and recovery computation times are dependent on #Shares, which is dependent on $r$ and $(l + b)$. The smaller the number of rows $r$, the fewer the number of shares (#Shares), and the lower are the computation times during secret sharing and secret recovery. But if $r$ is increased from $(l+b+1)$ to $(4b+l+c+1)$, from Inequality 3, the minimum number of servers required will decrease. Thus, the number of rows affects $\min(N)$ and the secret sharing and recovery computation times in opposing ways. For $l = 2$, $b = 2$, and $c = 2$, the tradeoff space is given in Table 7.

| r | min(N) | # Shares | Storage Blowup Per Server | Computation Time | |
|---|---|---|---|---|---|
| | | | | Secret Sharing | Secret Recovery |
| 5 | 45 | 5 | 1 | 333 $\mu$s | 165 $\mu$s |
| 6 | 30 | 15 | 5 | 1.103 ms | 490 $\mu$s |
| 7 | 21 | 35 | 15 | 2.668 ms | 1.150 ms |
| 8 | 24 | 70 | 35 | 5.480 ms | 3.020 ms |
| 9 | 18 | 126 | 70 | 10.31 ms | 6.276 ms |

**Table 7: Effect of increasing number of rows r on performance metrics when leakage-only fault threshold l = 2, Byzantine fault threshold b = 2, and crash fault threshold c = 2**

Table 7 shows that increasing the number of rows from $(l + b + 1)$ reduces the minimum number of servers required for that configuration while increasing the number of shares, #Shares, needed to store each secret. The storage capacity required at each server thus increases with $r$. Increasing #Shares will also increase the computation overheads at the users during the secret sharing and secret recovery processes. The practical range of $r$ is thus limited by the storage blowup and the computation overheads.

When there are five rows in the framework, each row gets a distinct share. The number of shares (#Shares) generated is minimum, and the computation times are small. But $45$ servers are required for this configuration. By having 7 rows in the framework, the minimum number of servers required is lowered by more than half to $21$ servers. For given fault tolerance and security thresholds, having fewer servers implies that a higher percentage of faulty servers is tolerated. Having fewer servers will also increase the manageability of the system. On the other hand, the storage blowup at each server increases by a factor of $15$. Since storage cost is cheap, this is a worthwhile tradeoff. The computation times are also at acceptable values when $r = 7$. Thus, the choice of the number of rows in the framework can be used to arrive at a suitable tradeoff point between the number of servers required, and the storage blowup and the secret sharing and recovery computation overheads.

### 7.3 Relation between Fault Tolerance and Security with Performance Given N Servers

In this section, we assume that $35$ data storage servers are available, and investigate the relation between the fault tolerance and security thresholds $l$, $b$, and $c$ and the performance metrics. We consider three cases. In each case,

we fix two of the thresholds at two servers, and increase the other threshold from one to five servers. Tables 8, 9, and 10 show the three different cases. For each combination of $(l, b, c)$, we fix the number of rows such that the secret recovery computation time is the smallest possible for the given configuration. Since the secret recovery computation time decreases with increasing $r$, for the given $(l, b, c)$, $r$ is set to the smallest value ($r \geq \frac{N(l+b)}{N-(3b+c+1)}$) such that $\min(N)$ is not more than 35 servers.

| l | r | min(N) | # Shares | Storage Blowup Per Server | Computation Time | |
|---|---|---|---|---|---|---|
| | | | | | Secret Sharing | Secret Recovery |
| 1 | 5 | 25 | 10 | 4 | 732 $\mu$s | 320 $\mu$s |
| 2 | 6 | 30 | 15 | 5 | 1.103 ms | 490 $\mu$s |
| 3 | 7 | 35 | 21 | 6 | 1.568 ms | 706 $\mu$s |
| 4 | 9 | 27 | 84 | 28 | 6.750 ms | 4.084 ms |
| 5 | 10 | 30 | 120 | 36 | 9.675 ms | 6.120 ms |

**Table 8: Effect of increasing the leakage-only fault threshold l on performance when Byzantine fault threshold b = 2, crash fault threshold c = 2, and min(N) $\leq$ 35 servers**

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

**Table 9: Effect of increasing the Byzantine fault threshold b on performance when leakage-only fault threshold l = 2, crash fault threshold c = 2, and min(N) $\leq$ 35 servers**

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

**Table 10: Effect of increasing the crash fault threshold c on performance when leakage-only fault threshold l = 2, Byzantine fault threshold b = 2, and min(N) $\leq$ 35 servers**

From Table 8, increasing the leakage-only fault threshold $l$ leads to a tolerable increase in the storage blowup per server, while the secret sharing and secret recovery computation times become high for $l \geq 4$ servers. The effect of increasing the Byzantine fault threshold $b$, as shown in Table 9, has a more adverse effect on the performance metrics. The storage blowup per server and the secret sharing and recovery computation times increase rapidly with increasing $b$. Thus, to achieve a very high performance with 35 servers, only a relatively small number of Byzantine failures can be tolerated.

On the other hand, the framework can accomodate more crash failures without any substantial performance impact, as shown in Table 10. Increasing the crash fault threshold from one to five servers leaves the performance metrics mostly unchanged. The storage blowup at each server is tolerable and the computation throughputs are maintained at acceptable levels.

The examples considered above demonstrate that the framework can tolerate crash failures with little performance impact, leakage-only faults with medium peformance impact, and a limited number of Byzantine faults. The maximum number of faults that can be tolerated is given by Equation 4. Thus, given $35$ servers, when $b = 2$ and $c = 2$, up to $24$ leakage-only faults can be tolerated; when $l = 2$ and $c = 2$, up to 7 Byzantine faults can be tolerated; and when $l = 2$ and $b = 2$, up to $24$ crash faults can be tolerated. However, practical limits on the secret sharing and recovery computation times and the storage blowup at each server are a more severe restriction on the actual range of faults that can be tolerated. Notice that, except for high values for the Byzantine fault threshold $b$, the secret sharing and recovery computation times are much smaller than the figures given for verifiable secret sharing in Table 2.

### 7.4 Relation between Fault Tolerance and Security with Performance Given Restriction on Secret Recovery Computation Time

Since increasing $l$, and $b$ in particular, can lead to a substantial increase in secret sharing and secret recovery computation times, as observed in Table 8 and Table 9, we remove the requirement of having only $35$ storage servers available, and instead impose the requirement that the secret recovery computation time for $8$ KB of data must be less than $1.6$ ms. The secret recovery computation time is important when reads are more frequent than writes, which is often the case. A secret recovery computation time of $1.6$ ms for $8$ KB of data is approximately six times and eight times slower than the decryption time using the Rijndael encryption algorithm for key sizes of $32$ bytes and $16$ bytes respectively, as was shown in Table 3.

| l | r | min(N) | # Shares | Storage Blowup Per Server | Computation Time | |
|---|---|---|---|---|---|---|
| | | | | | Secret Sharing | Secret Recovery |
| 1 | 6 | 18 | 20 | 10 | 1.494 ms | 660 $\mu$s |
| 2 | 7 | 21 | 35 | 15 | 2.668 ms | 1.150 ms |
| 3 | 7 | 35 | 21 | 6 | 1.568 ms | 706 $\mu$s |
| 4 | 8 | 40 | 28 | 7 | 2.109 ms | 928 $\mu$s |
| 5 | 9 | 45 | 36 | 8 | 2.742 ms | 1.196 ms |

**Table 11: Effect of increasing the leakage-only fault threshold l on performance when Byzantine fault threshold b = 2, crash fault threshold c = 2, and secret recovery computation time $\leq$ 1.6 ms**

| b | r | min(N) | # Shares | Storage Blowup Per Server | Computation Time | |
|---|---|---|---|---|---|---|
| | | | | | Secret Sharing | Secret Recovery |
| 1 | 6 | 12 | 20 | 10 | 1.494 ms | 415 $\mu$s |
| 2 | 7 | 21 | 35 | 15 | 2.668 ms | 1.150 ms |
| 3 | 7 | 42 | 21 | 6 | 1.568 ms | 1.00 ms |
| 4 | 8 | 64 | 28 | 7 | 2.109 ms | 1.60 ms |
| 5 | 8 | 144 | 8 | 1 | 592 $\mu$s | 576 $\mu$s |

**Table 12: Effect of increasing the Byzantine fault threshold b on performance when leakage-only fault threshold l = 2, crash fault threshold c = 2, and secret recovery computation time $\leq$ 1.6 ms**

Similar to Section 7.3, we consider three cases. In each case, we fix two of the fault thresholds at two servers, and increase the other fault threshold from one to five servers. Tables 11, 12, and 13 show the three different cases. For each combination of $(l, b, c)$, we fix the number of rows $r$ that gives the smallest $\min(N)$ while maintaining the secret recovery computation time to be less than $1.6$ ms. Restricting the secret recovery computation time limits the number of shares (#Shares) generated, which in turn keeps the storage blowup at each server reasonable. In

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

**Table 13: Effect of increasing the crash fault threshold c on performance when leakage-only fault threshold l = 2, Byzantine fault threshold b = 2, and secret recovery computation time $\leq$ 1.6 ms**

Table 11, the minimum number of servers required ($\min(N)$) shows a moderate increase with increasing $l$. When $l = 5$ servers, a total of 9 ($l + b + c$) servers out of 45 servers are faulty. That is, up to $20\%$ of the servers can be faulty (leakage-only, Byzantine, or crash), which should be acceptable. In Table 12, the minimum number of servers required ($\min(N)$) increases rapidly with the Byzantine fault threshold $b$. Thus, the proposed framework is suitable for tolerating a small number of Byzantine faults.

In Table 13, the computation throughputs and the storage blowup remain the same with increasing crash fault threshold $c$ for the example considered. With 21 servers, up to two crash faults are tolerated, and with 28 servers, up to 5 crash faults can be tolerated. Note that with 5 crash faults, a total of 9 servers out of 28 servers can be faulty. That is, up to $32\%$ of the servers can be faulty, which is a standard property of replica management protocols that tolerate only Byzantine faults. While in this example most of the faults are crash faults, the number of servers required is reasonable.

Thus, from Tables 11, 12, and 13, low secret recovery computation times can be achieved with acceptable requirements on the number of servers and the storage blowup at each server. As observed in Section 7.3, the requirement on the number of servers for tolerating crash and leakage-only faults is acceptable, while practical considerations will restrict the number of Byzantine faults that can be tolerated. Note that, in all the analyses, the number of rows in the framework was manipulated to arrive at the optimum configuration.

# 8 Discussion

This paper presents a novel approach for realizing a secure and fault tolerant data storage service in collaborative work environments. Key highlights of our work are:

- The use of perfect secret sharing for providing confidentiality of stored data eliminates the need for cryptographic keys used for encryption purposes, thus avoiding key management issues.
- Verifiable secret sharing schemes are typically used with perfect secret sharing schemes to achieve Byzantine fault tolerance. We show that verifiable secret sharing schemes incur substantial computation overheads, and are over 3000 times slower than the Rijndael encryption algorithm.
- We use XOR secret sharing for confidentiality, and manage each share using replication-based protocols for Byzantine and crash fault tolerance. The computation overheads are reduced drastically when compared to verifiable secret sharing schemes, but additional servers and storage capacities at each server are required. An example where the secret recovery computation time was only up to six to eight times slower than the Rijndael decryption algorithm was given.
- We present an architectural framework, called *GridSharing*, whose dimension can be varied to tradeoff between the number of servers required, and the storage blowup and secret sharing and recovery computation

times. This property was shown to be valuable in arriving at optimum configurations for different fault thresholds.

- We introduce a new fault model consisting of crash, Byzantine, and *leakage-only* faults for our analyses. We believe this new fault model will prove to be useful for analyzing works that are common to the areas of fault tolerance and security.
- For secret recovery computation times that are six to eight times slower than Rijndael decryption, we show that our proposed framework provides good fault tolerance to leakage-only and crash faults with acceptable overheads. However, in practice, resource limitations place a restriction on the number of Byzantine server failures that can be tolerated.

An important characteristic of the *GridSharing* framework is the tradeoff between the number of servers required and the storage blowup at each server. This tradeoff is worth considering because storage space is cheap, while more servers could result in manageability problems. In [24], the fact that storage space is cheap is exploited to keep data confidential. Encrypted data is encoded into shares using Shamir's scheme and embedded in a huge file. An adversary must obtain at least a large chunk of the file to obtain enough shares, and the transfer of such huge amounts of data over the network will result in the intrusion being detected. Likewise, we favor increasing the storage blowup at each server (to acceptable levels) while decreasing the number of required servers, as this will lead to easier intrusion detection and system administration.

Due to space considerations, the communication overheads during reads and writes, which will increase with the storage blowup at each server, have been omitted in our analyses. However, we note that a number of works have considered the problem of reducing communication overheads. Quorum systems [25] are replication-based techniques that can be used to balance the read and write communication overheads relative to each other by setting the quorum sizes appropriately. [26] shows that the use of cryptographic hashes can significantly reduce the communication overheads. [27] investigates the tradeoff between computation and communication overheads for several lossless compression algorithms. The use of cryptographic hashes and compression algorithms reduce communication overheads while increasing the computation overheads, which reinforces the need for reducing the computation overheads during the secret sharing and recovery processes.

# 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*, 1987.

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

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

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

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

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

[8] R. J. Anderson, "The eternity service," in *Proc. $1^{\text{st}}$ Intl. Conf. on Theory and Application of Cryptography*, 1996.

[9] A. Iyengar, R. Cahn, C. Jutla, and J. Garay, "Design and implementation of a secure distributed data repository," in *Proceedings of the $14^{\text{th}}$ IFIP International Information Security Conference*, 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," in *Proc. of the International 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 *Proceedings of the $14^{\text{th}}$ IEEE Symposium on Security and Privacy*, 1991.

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

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

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

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

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

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

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

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

[21] "The miracl software library." http://indigo.ie/˜mscott/.

[22] H. Krawczyk, "Distributed fingerprints and secure information dispersal," in *Proceedings of the $12^{\text{th}}$ ACM Symposium on Principles of Distributed Computing*, 1993.

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

[24] D. Dagon, W. Lee, and R. Lipton, "Protecting secret data from insider attacks," in *Proceedings of the $9^{\text{th}}$ International Conference on Financial Cryptography and Data Security*, 2005.

[25] D. K. Gifford, "Weighted voting for replicated data," in *Proc. of the $7^{\text{th}}$ Symp. on Operating Systems Principles*, 1979.

[26] D. Tulone, "Enhancing efficiency of Byzantine-tolerant coordination protocols via hash functions," in *Proceedings of the $10^{\text{th}}$ Euro-Par conference*, 2004.

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