Design of a Secure and Fault Tolerant Environment for Distributed Storage

Arnab Paul, Sameer Adhikari, Umakishore Ramachandran
College of Computing
Georgia Institute of Technology,
Atlanta, GA 30332, USA
{arnab, sameera, rama }@cc.gatech.edu

Abstract

We discuss the design and evaluation of a secure and fault tolerant storage infrastructure for un-trusted distributed computing environments. Previous designs of storage systems for this space have tended to use decoupled mechanisms for achieving fault tolerance and security. Our design, based on cryptographic properties of error-correction odes, combines redundancy (for fault tolerance) and encryption (for security) in a single unified framework. Our protocol can handle Byzantine faults and ensures confidentiality in a completely un-trusted environment. We qualitatively demonstrate the practicability of this approach. We also quantitative comparison of scheme and two other approaches, namely, Pure replication based techniques and SecureIDA scheme, and discuss their mertis and demerits.

1 Introduction

The growth in the volume of information handled by modern applications, the falling price of storage units, and the rapid improvement in network speeds have accelerated the research endeavor in distributed storage systems. These storage systems guarantee high availability of data in the presence of machine failures. The distribution of units can be at various levels; they could be geographically separated nodes connected via the Internet, or nodes distributed over a LAN, or even an array of disks in a RAID-like [6] architecture. Irrespective of the scale of distribution, the key principle that enables high availability (or fault tolerance) is the redundancy of information across different storage units. However, the degree and the specifics of the way in which redundancy is added, control the fault tolerance limits of the system.

In addition to fault tolerance, another very important issue that is critical in the design of such systems is security. Typically these systems are accessed by multiple users and are often connected to the internet and are thus potential targets for malicious attacks. While encryption is widely used to ensure the confidentiality of data, malicious parties can simply modify the data, which may go undetected causing potentially critical situations. Therefore, such systems need to have Byzantine fault tolerance to handle both hardware failures and data corruption, maintaining confidentiality at the same time.

Previous design approaches for distributed storage systems can be classified into two major categories, viz., (i) *Pure replication based methods* and (ii) *Transformation based techniques*. In a replication based strategy, data is replicated several times and then the replicas are stored on different servers (or storage units). In the retrieval phase, a certain number of these replicas are accessed and compared in order to obtain the original document. The overhead in this scheme is that of byte copying during the write phase and

that of the comparison cost during the read phase. However, this simple design leads to very high space complexity. Additionally, to ensure confidentiality of data, documents must be encrypted, thus adding the encryption/decryption costs to the write/read latencies.

A transformation based scheme can be viewed as a mapping from a lower dimensional space to higher dimensional one. A document of length m is inflated in size to length n $(n \geq m)$. The inflated document is now split into multiple pieces and each piece is stored on one of the storage units. The original document can be reconstructed even if some of the pieces are missing. Seminal work in this area by Rabin [26] showed how to design such a scheme to obtain a fault tolerant storage. Intuitively, such a scheme can also be modified to guarantee data confidentiality, provided no more than a specified maximum number of servers (storing the data pieces) ever collude to extract the data. It is also known that this scheme is in general highly space optimal, requiring minimal redundancy to enable a certain degree of availability. However, this scheme no longer remains secure while deployed over a completely untrusted set of servers, i.e., all of them are allowed to collude to extract the document. Encryption has to be added to safeguard the data increasing the associated access cost.

A variant of the transformation-based approach uses error correction codes (ECC). Designs for distributed storage systems based on error correction code have been proposed by some researchers [2, 32]. ECC based techniques provide redundancy in a space optimal way, leading to a space-optimal design for reliability.

In this thesis, we present SAFE, an ECC based scheme, that combines fault tolerance and encryption in a single set of operations. In particular we exploit cryptographic properties of some specific error correcting codes, such as generalized Reed-Solomon codes and Goppa codes [25], that allow us to use a single transformation that adds both redundancy and encryption to the data. One main feature of our system is the reduced key management overhead and therefore reduction in security risks. For large distributed storage, key-management is a significant problem, which is further aggravated by symmetric-key encryption. To avoid such glitches, different techniques have been proposed. Use of public-private key pairs for every participant reduces the key overhead; for using symmetric key, one needs to maintain a key for every writer-reader pair of data which leads to potentially $\mathcal{O}(n^2)$ keys to be maintained. Moreover, given that every participant is not equally secure, compromising the weaker parties leads to secret key being revealed. Public-private key pairs need only $\mathcal{O}(n)$ keys to be maintained, one pair for each participant. We use this to our advantage; the codes that are used to provide fault tolerance can also be exploited to provide security with almost no additional computational cost. The other properties of the proposed scheme are (1) fast writes, and (2) absolute data integrity. These properties are in line with two observations about storage systems.
*Dominance of writes over reads***:** In many secure distributed collaborations, there are many more writes(updates) than reads. Consider a standrad CVS application. Although the shared files are supposed to be accessed concurrently, typically there is little overlap between the work-hours of the individual users. However, the users keep checking in their local copies with every small update under the presumption that any other user should have access to the most recent version. Hence a single read is usually followed by multiple writes [12]. As another example, one can think of a smart home enabled with multiple sensors and data aggregators that capture and store information in a continuous fashion. However, only parts of them are typically analyzed at a later point of time depending on what needs to be analyzed. To this end, we note that Goppa codes provide *fast* joint encryption-replication. The read operation is comparatively slower to other alternatives. However, in a write-dominated system this design choice is a reasonable one.
Probabilistic guarantee of compare by Hashing Although cryptographic hashing has been accepted as a standard and unquestionable technique to verify data-integrity, the guarantee is only probabilistic. First, this may not simply be acceptable for certain critical data such as medical records. Second, despite the argument that hash-collision probabilities are less than probabilities of hardware faults, this argument is true

only for completely random inputs. Further there has been recent evidence that it may not be as risk free as commonly envisaged [9]. As we will see in later sections, some alternative design principles depend heavily on hashing while our design does not.

The contribution of this paper is two fold. First, we present the design for a Secure And Fault tolerant Environment for data storage (SAFE). Second we evaluate the performance of our system with respect to the alternatives. We do a comparative study of SAFE, a replication scheme augmented with encryption and Secure IDA. We believe that this is the first study of its kind.

The rest of the paper is organized as follows. Section 2 presents related work, and lays the the ground work for our proposal. Section 3 delves into the preliminaries of error correction codes and why this is relevant to fault tolerant storage. It also talks about Goppa codes [25], and their relevant properties. Section 4 discusses the design of SAFE. Section 5 presents the evaluation of our system: the methodology and the performance with respect to the alternatives. Section 6 presents the conclusions and the future work we intend to pursue.

2 Related Work

There are two primary directions of research in the space of distributed storage designs: Pure replication based and transformation (and fragmentation) based strategies. Quorum systems [3] have been used to provide coordination in distributed systems. Quorum approach is pure replication based. A quorum can be viewed as a collection of subsets over a universe of servers so that any pair of subsets satisfy certain intersection properties. Early works on quorum system considered how to handle benign failures [8, 31]. Byzantine failures, where the servers maliciously corrupt data, and collude among themselves, were studied later on [18, 22, 19]. The replication techniques studied in these investigations were adopted in the design of persistent object stores, such as Phalanx [20] and Fleet [21].

Another alternative to handle byzantine faults in a distributed environment is replicated state machine approach [27]. Castro and Liskov [5] presented a practical implementation based on this approach; they built a file system that handles byzantine faults. The key idea used in [5] is to replace public key operations by Message Authentication Codes that results in very small overhead. Overall, the replication schemes are not space optimal; to safeguard against f faulty servers, at least $3f + 1$ replicas need to be maintained [23]. Moreover, these schemes do not offer any inherent confidentiality for data; these schemes have to be augmented with encryption to assure confidentiality.

Transformation based approaches were initially designed to protect against benign failures. A very simple example is adding extra parity bits to the data in a RAID-like [6] system. In [26] Rabin presented an efficient Information Dispersal Algorithm (IDA) that can be used for fault tolerance in parallel and distributed systems. The scheme works as follows. Let n be the number of servers storing the data. Split the data into m pieces ($m < n$). Imagine each piece to be a vector of length m. By using a linear transformation (which can be thought of as an $n \times m$ matrix $T_{(n,m)}$), convert this vector into a vector of length n. Store each piece of this new vector in one of the servers. If the transformation can be suitably designed so that any m columns are linearly independent, then the original m vector can be reconstructed from any m pieces. Thus the scheme can tolerate up to $f = n - m$ failures and is provably space optimal. However, this scheme cannot guard against Byzantine faults as there is no way of knowing during retrieval if a data piece has been altered by the server.

Krawczyk [13] extended the IDA scheme to handle Byzantine faults, by appending fingerprints of each data piece along with the fingerprint of the entire content. Intuitively the scheme works as follows - first, with

the help of the fingerprints, the integrity of the data pieces can be verified, and once the required number of unaltered pieces are identified, the original document can be retrieved using the IDA scheme. This extension does not solve the security/confidentiality issue. However, the distributed fingerprinting can be combined with secret sharing [28] in a clever way that uses symmetric key encryption; the resulting scheme is shown to be secure with short secret sizes [14]. This approach, known as **SecureIDA** was exploited in the design of e-Vault, an electronic storage system developed at IBM [10]. A hybrid approach that combines secret sharing and replication based strategies has recently been developed by Lakshmanan et al [16]. This scheme tries to retain the best aspect of both the schemes and offer various levels of security guarantee, along with other flexibilities.

We conclude this section by mentioning a few other distributed storage systems that have been reported recently. The PASIS architecture developed at CMU provides a combination of decentralization, redundancy and encoding along with dynamic self-maintenance in the design of a *survivable information storage* [33]. The OceanStore project at Berkeley is a global scale information system designed with a goal to be able to supply data anywhere and anytime and therefore combines decentralization and cryptographic techniques in its architcture [15]. Farsite [1] is a scalable file system developed at Microsoft Research, that provides the abstraction of a centralized file system over a set of physically distributed untrusted workstations acting as storage units.

3 Error Correction and Fault Tolerance

3.1 Preliminaries

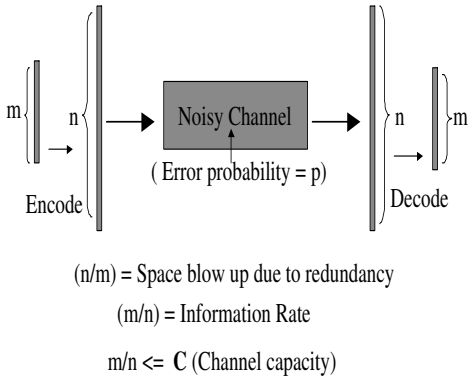

Figure 1. Shannon's Observation on Information Rate over a Noisy Channel

In this section we try to draw a connection between the theory of error correction codes (ECC) and the design of space efficient fault tolerant storage. ECC has been studied in numerous contexts, and chiefly in connection with the transmission of messages over noisy communication channels. Figure 1 shows such a scenario. The message to be transmitted is of length m. However, because of the noise in the channel, some of the bits are modified with some error probability p. Notice, that this error probability is an intrinsic property of the channel and serves as an abstraction of the physical characteristics of the channel that gives rise to this transmission noise. At this point one can see a clear analog between a noisy channel and a failure prone storage; the writer in this case has the role of the sender and the reader acts like the receiver. To safeguard against the errors in the channel (or the storage), one can add redundancy to the message so that even if some of the bits are corrupted the original message can be recovered. In Figure 1, the original message (of length m) is inflated with redundancy bits to length n and then transmitted over the channel.

The quantity m/n is known as the *Information Rate*, since this defines what fraction of the total transport is the original information content. What is the theoretical upper limit of Information Rate? In his classic 1948 paper that opened the field of modern communication theory, Shannon showed that for any channel, there exists a quantity called the *Channel capacity* (C), that serves as the upper bound of the information rate [29]. In our case, the failure probabilities of the storage units abstractly define this quantity C; given an accurate estimate of this probability, the upper bound can be determined. However, for all practical purposes, one can replace the probabilities with the expected number of errors (f) and thence design all subsequent algorithms.

Shannon's paper that had mostly information theoretic ideas, did not have any constructive proof that the bound C can indeed be attained; the proof was existential in nature. One primary goal of the theory of ECC is to investigate how close to this limit the information rate can be pushed by explicit algorithms. Therefore, it is quite natural that one would look into ECC techniques to design space-optimal redundancy algorithms to build fault tolerant storage. There are numerous error correction schemes with different Information Rates. The algorithmic complexity increases as the Information rate is improved. For a clear understanding of Information and ECC theory, the reader can refer to texts such as [7] and [25], respectively. For our purposes, we have focused on a specific ECC, viz., Goppa codes [25]. These codes, in addition to having good information rates, offer certain cryptographic properties that can be exploited in designing secure and fault tolerant storage.

Reed-Solomon and Goppa Codes

Next, we present the construction of Reed-Solomon and Goppa codes in brief, since a detailed mathematical description is beyond the scope of this paper and perhaps not as much relevant as the main design principle. The main intent is to bring out the cryptographic properties of these codes that make them attractive for designing storage systems for untrusted environments. e A linear (n, k, d) code ($n \geq k$) is a linear mapping E from a set of strings of length k to the strings of length n, such that for any two strings x, y (of length k) the Hamming distance between the encoded strings, $D(E(x), E(y)) \geq d$. This means any two strings of length k are now separated by at least a distance d in the encoded form. For any linear code, there is an associated *generator* matrix G (dimension $n \times k$) that maps strings (vectors) of length k to strings of length n. Encoding a string therefore is simply multiplying the vector with the Generator matrix. On the other hand, checking whether a string of length n is really a codeword involves another matrix H (dimension $(n-k) \times n$), known as the *parity check matrix*. The rows of H are such that they give a basis to the null space of G, i.e., $G.H^T$ is a zero matrix. Any codeword $\alpha = Gx$ therefore also belongs to the null space of H, giving $\alpha H^T = 0$. This is how one checks if a transmitted packet is a codeword or not. If $\alpha H^T \neq 0$, then it is an indication that the transmitted packet has been corrupted. A linear code is completely defined with the help of its generator and parity matrices.

A Reed-Solomon code is a specific class of linear code. The easiest way to think about such a code is as follows. Consider some finite field F_q of order q and imagine a polynomial $f(x)$ of degree $k < q$, over the field, i.e., $f(x) = f_0 + f_1 x + \ldots + f_{k-1} x^{k-1} \mid f_i \in F_q$. Let $\alpha_0, \alpha_1, \ldots \alpha_{n-1}$ be n points in the field ($k < n < q$). Consider the vector obtained by evaluating $f(x)$ in all these points, i.e., $\hat{f} = [f(\alpha_0), f(\alpha_1), \ldots f(\alpha_{n-1})]$. This vector $\hat{f}$ can be thought of as the encoding of the message block $[f_0, f_1, \ldots f_{k-1}]$. If $g(x)$ is a different polynomial of degree k, the corresponding vector $\hat{g}$ can agree with $\hat{f}$ on at most $k-1$ points, since two polynomials of degree at most $k-1$ can share at most $k-1$ roots. Hence the encoded vectors differ in at least $n-(k-1)$ points, which is the minimum distance of this code. One can easily show that this polynomial formalism can be easily cast in the form of a linear transformation.

A Goppa code is another kind of linear code that is determined by an irreducible polynomial $g(x)$.

$$g(x) = g_0 + g_1 x + g_2 x^2 + \ldots g_r x^r, \quad g_r \neq 0$$

The coefficients and the variable x can take values from a field $F_q|q = p^m$ for some prime number p and integer m. Let $\alpha = (\alpha_1, \alpha_2, \ldots \alpha_n)$, $\alpha_i \in GF(q)$, such that for all j, $g(\alpha_j) \neq 0$. Let $\hat{t} = (t_1, t_2, \ldots, t_n), t_i \in GF(p)$. A Goppa code is a collection of vectors $\hat{t}$ that satisfy the following equation.

$$\Sigma_{j=1}^{n} \frac{t_j}{x - \alpha_j} = 0 \mod g(x)$$

A Goppa code is completely determined by the Goppa polynomial $g(x)$.

Goppa codes have fast decoding algorithms, in particular they can be decoded using known algorithms to decode other standard codes, viz., the Generalized Reed Solomon Codes. For example, the well known Peterson Algorithm for GRS codes can be adopted to decode Goppa codes. The decoding in such case is of order $O(f^3 + nf)$, where f is the number of bit errors in a binary string. In what follows, we describe the cryptographic properties of these codes.

3.2 Cryptographic Properties of Linear Codes

McEliece Cryptosystem

Decoding a general binary linear code is NP-complete [4]. Based on a similar intuition McEliece developed a public key cryptosystem that exploits the hardness of syndrome decoding of a Goppa code. In brief, the scheme works as follows. A private key consists of four items: (i) an irreducible polynomial, the Goppa polynomial, (ii) a permutation of n elements $\alpha = (\alpha_1, \alpha_2, \ldots \alpha_n)$, $\alpha_i \in GF(q)$, (iii) a parity check matrix H (this has to be computed) and (iv) a matrix S that scrambles the plaintext.
The public key consists of the generator matrix G computed in a particular way; we skip the details of this generation process for brevity.

To encrypt a plain text s, a sender gets hold of the public key, computes the ciphertext as $c = s.G + e$. Where e is any random error pattern up to f bits, where f is the maximum number of errors the code is designed to correct. The private key holder, on having the parity check matrix H, will be able to recover a codeword even in the presence of f errors or less. However, an adversary will have to face the decoding problem without knowing the error vector e. This hardness is the basis of McEliece Cryptosystem [24]. To date all known attacks on this scheme are exponential; there is no known subexponential structural attack that might distinguish between a permuted Goppa code from a random code. Just as a contrast, the hardness of decoding RSA, the most popular public key infrastructure, is not known to be NP-complete and subexponential structural attacks are also known to exist.

We exploit this cryptographic property of these linear codes. When combined with the default fault tolerance feature, we believe these codes provide a powerful design principle for secure fault tolerant storage. In the next section we describe how exactly we use linear code based encoding/encryption in our storage design.

4 Design of SAFE

Figure 2 shows the block diagram of a distributed data store. There is an un-trusted storage medium. A producer of the data, denoted as **W** writes onto this storage. Subsequently the data is read by a consumer, **R**. Both **W** and **R** are potentially trusted parties. However, the nodes where the data is stored can be completely

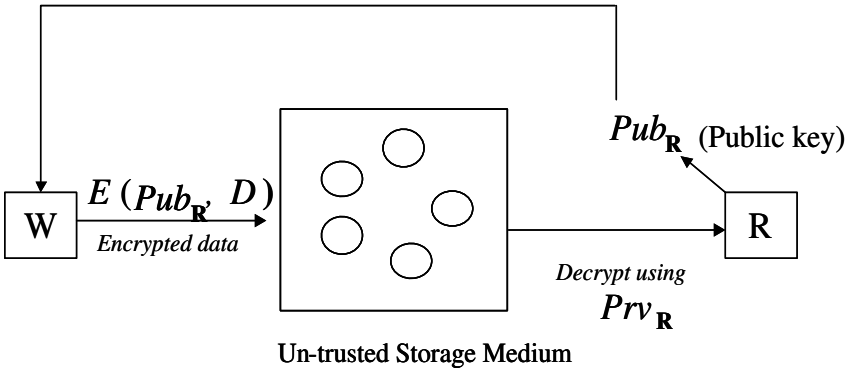

Figure 2. Abstract view of a Distributed Secure Storage

un-trusted. The system is prone to Byzantine failures. We assume that at any given time, no more than f servers can modify the data maliciously. However, we do not assume any upper bound on the number of servers that may work together in order to read the data [1]. Clearly the threshold cryptographic schemes [11], that make an assumption about the number of colluding servers, will not work in such a scenario. To write a document, *W* grabs the public key published by *R*, encrypts it using this public key and then writes the encrypted message onto the medium. In this case the public keys are actually generated by the linear error correcting codes that we have discussed in the previous section. The first advantage of this scheme is the dual purpose served by the encoding operation; first as a means to provide fault tolerance and second as a strong encryption method. The absence of any symmetric key offers two more features. Neither the writer or the reader need to maintain pairwise symmetric keys. And next, the security risk is minimized; although we assume that both writer and reader to be trusted, for the kind of application class we consider, the writers (that could potentially be sensors generating volumes of data) could be compromised easily, for several reasons such as exposition to the external world, mobility etc.. In such cases, a compromised writer, which had access to a symmetric key would automatically lead to the violation of confidentiality. On the other hand, it is relatively safer to assume that the consumer of the data stay un-compromised for a much longer period of time. Since the encryption is done using the public key of this consumer, only a private key, which is held by only one party, can decrypt the data.

We assume that all writes for any single data block are strictly serialized. This is not a very restrictive assumption, because, even if there are multiple writers of a single block, the write requests can always be linearized at some location that may act as a gateway to these servers. This is similar to the assumption that multiple writers coordinate their write operations when they access the same file. For any write and read operation, the user tries to contact a server, up to a predetermined timeout. If the server does respond, communication takes place, otherwise, the part of the data being exchanged with this particular server is assumed to be erroneous, both for write and read operations. Use of timeout leaves the possibility of not being able to differentiate between the *slow* and the *faulty*. However, in a practical deployment, it should be reasonable to determine a timeout period long enough to filter out the non-responding units.

Figure 3 is showing the details in our scheme. The original data to be stored is m bits long; m is assumed to be a multiple of 8 and therefore it is essentially a $m/8$ byte long data block. Once these m bits are encoded to a string of n bits using a Goppa code, the bits are distributed and are stored on different servers.

[1]There are two aspects of security to consider: (i) *integrity*, for which theoretically there has to be an upper bound on the number of bit alterations, and (ii) *confidentiality* which can be independent of how many parties trying to decipher a cyphertext

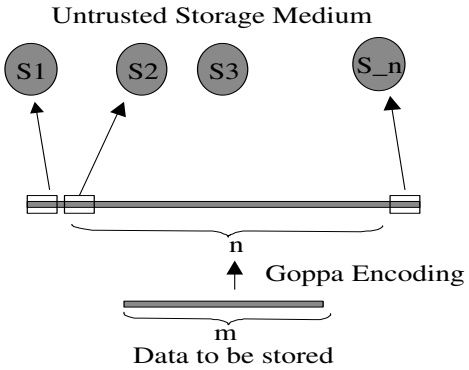

Figure 3. A Goppa Encoded Data block distributed to n Servers

The reason we have to consider the block as a string of bits (and not bytes) is because we are using a binary Goppa code that safeguards against bit errors. Handling done at a byte level will keep the bits all together, and thereby a single server failure will affect eight bits at a time. Such bulk errors can be dealt with by using specialized codes known as Erasure codes. However, in the scope of this paper, we did not consider such options and limited ourself to simply using Goppa codes. The bit level operation can also be averted if we used non binary Goppa codes that operate with a larger alphabet of symbols; such codes can perhaps consider a single byte as one symbol. However, again we limited ourselves to the binary design, because normally the binary codes are known to have better fault tolerance parameters.

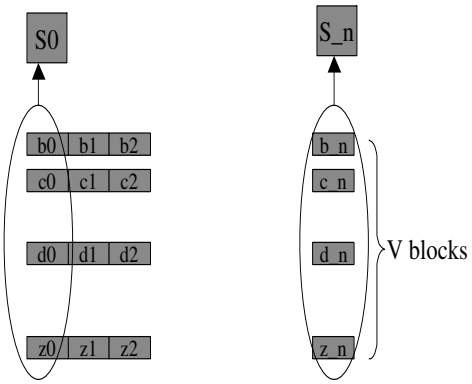

Figure 4. Bunching up V blocks to extract bit columns

Storing a single bit at a time on a server would be inefficient. In reality we bunch up a number of blocks that need to be stored and then isolate out the bits from these blocks, put these bits into a new block of writable length, and store the new block on to one server. Figure 4 describes this process. A number (V) of n length blocks of a file that is being written are bunched up. As seen in the figure the bit 0 of each block goes to server 0, bit 1 to server 1, and so on. Note that these blocks are output of a Goppa encoding module. If we take V to be a multiple of 8, it yields a block of $V/8$ bytes. Thus in one time we write $V/8$ bytes to each server. For small V, the write process is inefficient due to the transport overhead associated with writing to a server. The scheme therefore definitely yield better results in storing high volume of data at one time than small volumes. Meta-information, such as file name, block identifier and bit column identifier are

all appended along with the data blocks finally being stored. These informations are essential for a subsequent read phase.

Data Retrieval from the system is exactly the reverse of the write operation. The blocks retrieved from a server are essentially the bit columns of the original data. Once all such columns are acquired and arranged with the help of the meta information, the whole bit matrix is transposed to yield the original blocks. If some server does not respond, i.e., if some of the bit columns are missing, we fill them with zeros. These are the erroneous bits, and the decoding module in the very next step can extract out the data given by the user even in presence of these erroneous bits. Also notice that the presence of malicious servers (up to f of them) that may modify the data blocks, does not change anything. Because the ECC naturally handles any alteration (less than or equal to f) in that data that may happen for any reason. This is the primary distinction between supporting a fail-stop system and a Byzantine system. However, a Byzantine failure may involve one further complication; one malicious party masquerading as other parties and sending wrong data on everyone's behalf. Normally t is impossible to attend any distributed consensus even in the presence of a single malicious party, however, for *practical* fault tolerance it suffices to use authentication as a protection against such masquerading. One can use a standard public key infrastructure for this purpose, but that incurs performance penalty. A faster and cheaper alternative is to use Message Authentication Code (MAC) [5]. Any such mechanism can be integrated into our architecture without any incomapatibility factor and incurring only performance cost.

A user, while writing in this setting follows what we already described in section 3.2. He encodes the data block using Goppa codes and randomly adds t bit errors ($t \leq f$) to the block. An adversary trying to read this encrypted data, will have to solve the Syndrome Decoding problem for Goppa codes, which is provably hard. While, this takes care of confidentiality, the system is still robust to tolerate up to $f - t$ failures (because the user has already used up t errors to confuse the adversary). This enables the user to use the scheme in a public key infrastructure. However, when the storage is used just like a private storage, and no public key exchange is necessary, the Goppa code parameters need not be published at all. In that case it is not even required by the user to add the random error pattern, and thereby utilizing the full robustness of the system, i.e., tolerating up to f failures.

Implementation

The system is right now implemented as a user level library and is tested to work on Linux operating system. A gateway mediates between any client and these servers. The communication between the client and the gateway is assumed to take place through a secure channel. The servers all run a listener thread; a thread that listens for read and write commands from the gateway, the gateway receives the commands from the client. The gateway is assumed to be trusted, as it performs all the cryptographic operations with the plaintext. On receiving the plaintext from the client, the gateway performs encoding/cryptographic operations, transposes the bit arrays as we discussed before, generates the meta-data about the blocks and store the blocks (along with the meta data) on the servers. The read operations are done exactly in the reverse order. Note that such a gateway is not an absolute requirement in the architecture; all the tasks could as well be carried out by the client. The current implementation uses the gateway for design simplicity on the client end.

5 Evaluation

5.1 Objectives

In this section we evaluate SAFE on simple microbenchmarks. We also compare it with other two approaches discussed in this paper. To this end we used the same infrastructure that we built for developing

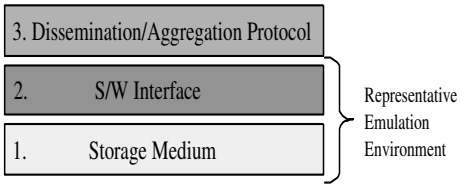

Figure 5. Layered view of Functionalities

SAFE, to evaluate the other two schemes as well. Recall that the view of secure distributed storage in Figure 2 is quite generic in nature. It can be used to represent either of SAFE, Secure IDA or replication with encryption. In fact, the design of eVault, the SecureIDA based system from IBM [10] has a similar architecture. The replication based schemes can also be faithfully reproduced in this setting. The immediate purpose of our study is not the design and implementation of a file system end to end, rather we wanted to asses the practicability of our scheme outside the shell of asymptotic order complexities. We also wanted to find out how well it compares with other techniques.

Figure 5 describes the typical cross section of a distributed storage. Layer 1 is the physical storage medium; it could be a disk array like RAID [6], or a bunch of virtual disks like Petal [17], or a host of servers with disks distributed over local or wide area network. Layer 2 is the software layer; depending on layer 1, it can be a driver for disk arrays, or a distributed file system like Frangipani [30], or simple nfs, or some wide area network file system respectively. The core protocol of a distributed secure storage is typically implemented above the first two layers. Our generic infrastructure provides a simple emulation environment spanning a bunch of servers over an nfs and implements the protocols corresponding to different storage schemes. We believe this approach serves two purposes for us. First, we can test the applicability of a Goppa code based algorithm for storage in terms of real life tractability of space and time orders. And second, we get a single platform to normalize the implementation of three different strategies to carry out a comparative analysis.

We have run all the experiments on a cluster of 16 nodes, each being an 8-way SMP, 550 MHz Pentium II, with 4GB RAM and 2MB L2 cache. The nodes run RedHat Linux 7.1. In all the experiments, we eliminated the client that we mentioned in the previous section, because the communication between the client and the gateway is simply a secure transmission of read and write data and it really does not involve any of the distributed protocols. Similarly, during the write and read operations, at the server end, the interaction between the nfs and the server thread executing the read/write command is beyond the scope of the protocols. Including this interaction with the disk, which is shared by many different users may perturb the actual reading for the cost of the protocols. To estimate this cost accurately, we account for the encoding and decoding costs, the cost of handling the meta information, and the communication overheads.

5.2 Comparison Platform

We have tried to reproduce the two schemes other than SAFE in a minimal but faithful way. For a replication based strategy, we felt that implementing a complete system, such as a quorum of a particular kind is not necessary, as that would include building mechanisms to handle consistency which we are not dealing in the first phase of SAFE's development. We emulate a write operation in a pure replication strategy in following steps - (i) encrypt the data using symmetric encryption; we used AES in particular, (ii) make $(3f+1)$ copies of this data and (iii) disseminate the copies among $3f+1$ servers. A read operation similarly

consists of - (i) reading the data back from at least $f + 1$ servers, (ii) take a majority voting to determine the correct copy of the data and then decrypt using the same key.
For the SecureIDA scheme, we take the following steps - (i) Inflate an m byte data into n bytes using a linear mapping with a row rank at least m, let's call these pieces d_i for ($1 \leq i \leq n$), (ii) with each d_i, append the vector $(H_1, H_2, \ldots H_i \ldots H_n)$, H_i being the hash digest of d_i (iii) Store each of these document pieces with a different server, n servers in all. The read operation retrieves the stored pieces, verifies from the hash digests the integrity of each piece and then choose from these pieces m uncorrupted pieces and assemble them to recover the original block of length m. The original scheme proposed by Krawczyk, does not require the entire digest vector to be appended to every piece. Instead, the digest vector can be processed with an ECC beforehand and only the pieces from this encoded vector be appended with each data piece. We avoided that complication here at the cost of slight blow up in space.

5.3 Results Summary

To implement SAFE we chose a Goppa code implementation with the following parameters. It takes blocks of 82 bytes and transforms them into blocks of 128 bytes, i.e., $n = 1024$ bits. As we discussed in section 4, we bunch up a number of blocks of size 82 and encode them into 128 byte blocks. This particular implementation can correct up to 37 errors [2]. To keep consistency with the experimental set up for SAFE, we choose similar parameters for other experiments as well, i.e, we consider blocks of 82 bytes; for SecureIDA scheme we transform them into blocks of 128 bytes, which gives a fault tolerance of 46. In the case of replication, no transformation of individual blocks is required, but it has the maximum space requirements. In fact, to safeguard against $f = 37$ errors, it has to make at least $3f + 1 = 112$ copies of each block and disseminate all of them.

Figure 6 summarizes the write latencies for SecureIDA and SAFE schemes as a function of number of bytes of data (expressed as the number of blocks of 82 bytes). The cost of pure replication based strategy was significantly high compared to the other two schemes; in order for a clear visual comparison between the performance of SAFE and SecureIDA, we choose to present only the latency for those two schemes. We see that SAFE latency is lower for most of the data points. For example, for 512 blocks, SAFE latency is about 1.6 seconds compared to 3.3 seconds taken by SecureIDA. Just as point of refrence, for the same number of blocks the pure replication scheme took more than 200 seconds. This is due to the fact that Goppa code has fast encoding. Note that in these experiments, SAFE and SecureIDA schemes are forced to handle the data in blocks, because of their algorithmic requirements. While there is no such algorithmic restriction for the replication scheme. It can ideally write any number of bytes in one operation. Though in reality this might not hold. For example, these schemes can as well be running on top of a disk array. In this case it is natural to expect that data transfer is limited to blocks of a maximum length. Just to emulate this behavior we imposed a restriction on the block length that could be written in one operation. In this case for the replication based scheme; for 512 blocks, the latency increased to 337 seconds, showing that this is perhaps unsuitable for at least disk array like environments.

Figure 7 notes the corresponding read latencies. In this case, the SAFE's performance was worse compared to the other two. The replication based scheme has a much improved latency for reading compared to writing, because on the average it had to read less number of replicas to find a majority and hence select one of them as the correct one. However, the latency worsened in the case of SAFE. For example, for 1536 block of data, SAFE read takes over 12 seconds, as opposed to 9 sec and 3.9 sec for the replication and SecureIDA schemes respectively. This is because Goppa module in the presence of failures has to do some

[2]Since the error correction codes are of combinatorial nature, they only come in discrete parameters that cannot be arbitrarily changed. The block lengths 82 and 128 in this case are the specific parameters that are allowed by Goppa Codes.

extra work called syndrome decoding.

Figures 8 and 9 shows the bandwidth offered by these different schemes. For write, the replication bandwidth is very low compared to the other two. We observe that SAFE and SecureIDA are comparable on this metric; while SAFE has a slightly declining slope, SecureIDA shows a rather rising curve (except at the end where it bends down slightly). Upto 2048 blocks, SAFE has higher bandwidth than SecureIDA. The reason for such decline in bandwidth is the bit-level block isolation that is necessary for SAFE (ref. figure 3). Implementing SecureIDA also needed similar isolation of vertical data blocks, however that is done at the byte level and hence much faster compared to bit level operation. Nevertheless, the two schemes are comparable for almost all data points. For read operation however, the bandwidth offered by SAFE is much worse compared to the other two schemes. However, it is reasonable to expect that in a system which is dominated by writes and critical integrity and confidentiality needs, a user may be willing to pay such prices for occasional reads.

We end this section on a general comparative note. Table 1 summarizes qualitatively different aspects of the three schemes we studied here. On the space optimality issue, replication is clearly very inefficient compared to the other two. SecureIDA, although has very good performance numbers on the data set used in this work, suffers from one weakness; lack of deterministic guarantee for data integrity. This is because, the integrity is maintained with the help of fingerprinting. It is possible, although with small probability that an attacker determines how to break a cryptographic hash function. In that case the SecureIDA will not be able to tolerate even a single Byzantine failure. The probabilistic guarantee for data integrity may not be acceptable for very critical documents and scenarios.

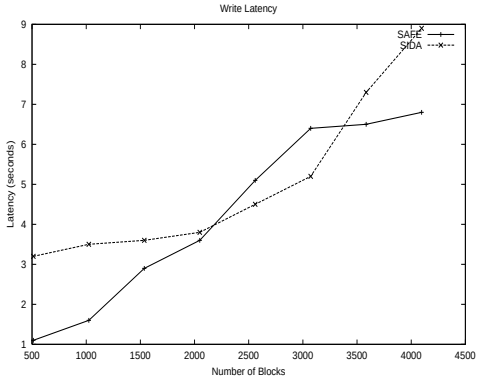

Figure 6. Write Latencies under different schemes

Properties	Scheme		
	SIDA	Rep	SAFE
Space	Good	Bad	Good
Integrity	P	D	D
Encryption	Reqd.	Reqd.	Not reqd.

Table 1. Qualitative Comparison, (P = Probabilistic guarantee, D = Deterministic guarantee)

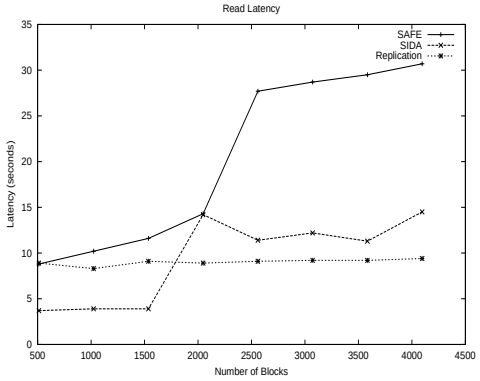

Figure 7. Read Latencies under different schemes

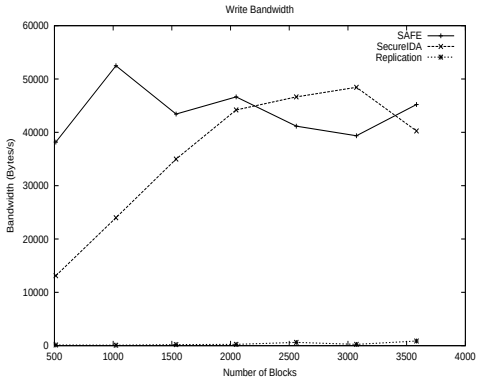

Figure 8. Bandwidth consumed in write under different schemes

6 Conclusions and Future Work

We discussed a new design principle, i.e, that of using Goppa codes based approach for building secure, fault tolerant storage. The design of SAFE exploits this principle. We demonstrated the practicability of this approach where most of the data transfer happens in bulk. We compared the SAFE approach with two other alternatives. Both SAFE and SecueIDA beat the replication approach in terms of space blow up. Although it appears from the results that the transformation or ECC based schemes are much better than pure replication schemes, this is only true for the selective case where the data transfer happens in bulk. Many applications may as well demand small blocks of data be read or modified. In such a case, reaching out to a large number (say a thousand) of servers , with a small number of bytes every time, may prove very inefficient. Which was predominantly believed to be major weakness of fragmentation based approach.

The design of SAFE serves as a launching pad for the development of an end to end file system, which is currently being investigated at our research group. Although practicability of the scheme is proven, we believe a further optimized set of Goppa Code algorithms can be designed. Among the drawbacks of this approach is the requirement of a large number of servers, which is equal to the number of bits in an encoded block. However, this can be dealt with if non binary codes of good parameters exist.

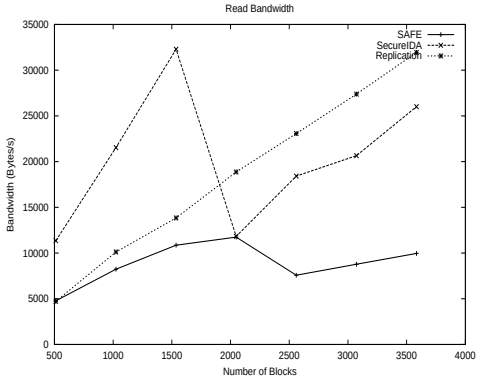

Figure 9. Bandwidth consumed in Read under different schemes

References

[1] A. Adya and et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation(OSDI)*, 2002.

[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern. Scalable secure storage when half the system is faulty. In *Automata, Languages and Programming*, pages 576–587, 2000.

[3] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65(5):223–228, 1998.

[4] E. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg. On inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, IT-24:384–386, 1978.

[5] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[7] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley and Sons Inc., 1991.

[8] D. Gifford. Weighted voting for replicated data. In *Proc. of ACM Symposium on Operating Systems Principles*, 1979.

[9] V. Henson. An analysis of compare-by-hash. In *Hot Topics in Operating Systems(HotOS)*, 2003.

[10] A. Iyengar, R. Cahn, J. Garay, and C. Jutla. Design and implementation of a secure distributed data repository. 1998.

[11] W. Jackson, K. Martin, and C. O'Keefe. Multithreshold secret scheme. *Advances in Cryptology (CRYPTO 93)*, 773:126–135, 1994.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[13] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc. 13th ACM Symp. on Principles of Distributed Computating*, pages 207–218, 1993.

[14] H. Krawczyk. Secret sharing made short. *Advances in Cryptology (CRYPTO)*, 773:136–146, 1994.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, 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 ACM ASPLOS*. ACM, November 2000.

[16] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. In *Proc. of 23rd International Conference on Distributed System (ICDCS)(to be held)*, 2003.

[17] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[18] D. Malkhi and M. Reiter. Byzantine quorum systems. pages 569–578, 1997.

[19] D. Malkhi, M. Reiter, and A. Wool. Optimal byzantine quorum systems. Technical Report 97-10, 17, 1997.

[20] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Symposium on Reliable Distributed Systems*, pages 51–58, 1998.

[21] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *The 2nd DARPA Information Survivability Conference and Exposition*.

[22] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of byzantine quorum systems. In *Symposium on Principles of Distributed Computing*, pages 249–257, 1997.

[23] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *16th International Symposium on Distributed Computing (DISC 2002)*, pages 311–326, 2002.

[24] R. J. McEliece. A public key cryptosystem based on algebraic coding theory. *Jet Propulsion Lab, DSN Progress Report*, 42(44):114–116, Jan-Feb 1978.

[25] O. Pretzel. *Error Correcting Codes and Finite Fields*. Clarendon Press, Oxford, 1992.

[26] M. Rabin. The efficient dispersal of information for security, load balancing, and fault tolerance. *JACM*, 36(5):335–348, April 1989.

[27] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[28] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[29] C. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–423, 1948.

[30] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[31] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. In *Database Systems*, volume 4, pages 180–209, 1979.

[32] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems: First International Workshop (IPTPS)*, 2002.

[33] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable information storage systems. *Computer*, 33(8):61–68, 2000.