Designing Secure and Robust Distributed and Pervasive Systems with Error Correcting Codes

A Thesis
Presented to
The Academic Faculty

by

Arnab Paul

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
February 10th, 2005
Designing Secure and Robust Distributed and Pervasive Systems with Error Correcting Codes

Approved by:

Dr. Umakishore Ramachandran, Advisor  
College of Computing,  
Georgia Institute of Technology

Dr. Mustaque Ahamad,  
College of Computing,  
Georgia Institute of Technology

Dr. Richard Lipton,  
College of Computing,  
Georgia Institute of Technology

Dr. Karsten Schwan,  
College of Computing,  
Georgia Institute of Technology

Dr. Ajay Mohindra,  
IBM Thomas J. Watson Research Center,  
Yorktown Heights, New York

Date Approved: February 10th, 2005
ACKNOWLEDGEMENTS

I express my gratitude to the committee members who evaluated this dissertation. They helped me in multiple ways on more than one occasions, on topics related to and beyond this thesis. I sincerely thank Mustaque, Karsten, Dick and Ajay for their cooperation.

The days at Georgia Tech as a graduate student have been wonderful. I am indebted to the kind help and cooperation that I received from all the staff members. I would like to especially mention Barbara Durham, Cathy Dunnahoo, and Barbara Binder. The administrative staff members at CERCS became friends, especially Jennifer, Deborah and Susie. I'll always have nice memories of an excellent environment they provided us in terms of every cooperation possible. I'd also like to acknowledge Dani Denton for the wonderfully sweet lady she has always been.

My friends at GaTech and in the city of Atlanta, always had a very special part in my life. I sincerely extend my gratitude to them for being able to share great moments and time. I would like to specially mention Angshuman, Sameer, Bikash, Vikram, Srikanth, Alok, Shekhar, Kavita, Susendar and many others. I extend my sincere gratitudes to Sandip for being such a lovely roommate and colleague. All members of our Ubiquitous computing group deserve extra special mention. We built things together, wrote papers, shared acceptance as well as rejections; the bond only grew stronger. Phil Hutto, Matt Wolenetz, Namgeun Jeong, Hasnain Mandviwala and Nissim Harel deserve special mention. Minaxi Gupta, another fellow graduate student, and now a professor, had been a sweet and caring and helpful friend that I will always remember. Ajit Motkur has been the buddy that can never be matched.

Some of the friends became more than part of life, and I cannot begin to describe my gratitude for them. Sandip Ray at Austin has been a great moral support during the moments of unbearable despair. Rahul Ray always helped me discover myself; what more to want from a friend? I shared perhaps the most amazing moments of friendship with Gaurav;
we did things together and continue to share our dreams that revitalized the meaning of life for me time and again. Rajnish Kumar has been more than a younger brother to me, from whom I learnt how to make myself a better person.

My last but the deepest thanks and regards are reserved for my advisor, Kishore. What can I say about a person whom I hold as my idol? From Kishore, I had lessons that reached far beyond a dissertation; I learnt from him how to face life with a smiling face and attitude. Kishore is incredible and I owe every bit of my doctoral career to him.

I acknowledge some of the magic moments that made my stay in this city a most memorable one to a very special person in my life - Aparna.

And I dedicate this thesis to my mother; without her I would be nothing.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS .................................................. iii

LIST OF TABLES ........................................................ vii

LIST OF FIGURES ...................................................... viii

SUMMARY ................................................................. ix

CHAPTER 1 INTRODUCTION: AN OVERVIEW ......................... 1
  1.1 Problem Statement .............................................. 1
  1.2 Large scale Distributed Storage ............................. 2
  1.3 Remote Authentication ........................................ 2

CHAPTER 2 A SINGLE ARCHITECTURE FOR FAULT TOLERANCE
  AND SECURITY ....................................................... 4
  2.1 Introduction .................................................... 4
  2.2 Related Work ................................................... 8
  2.3 Error Correction and Fault Tolerance ...................... 10
     2.3.1 Preliminaries .......................................... 10
     2.3.2 Cryptographic Properties of Linear Codes .......... 12
  2.4 Design of SAFE ............................................... 13
  2.5 Evaluation ..................................................... 19
     2.5.1 Objectives .............................................. 19
     2.5.2 Comparison Platform .................................. 20
     2.5.3 Results Summary ....................................... 21
  2.6 Discussions ................................................... 23

CHAPTER 3 AN EXTENSIBLE STORAGE ................................. 25
  3.1 Introduction ................................................... 25
     3.1.1 Motivation and Challenges .......................... 26
     3.1.2 e-SAFE ................................................ 29
  3.2 System Overview .............................................. 30
     3.2.1 Design Overview ....................................... 30
     3.2.2 Design Rationale ...................................... 31
3.3 Architecture .................................................. 35
3.3.1 Erasure Coding ............................................ 39
3.3.2 Verification .................................................. 40
3.4 Implementation ................................................. 41
3.5 Evaluation ..................................................... 42
  3.5.1 Microbenchmarks .......................................... 43
  3.5.2 Performance under Workload ............................ 46
3.6 Related Work .................................................. 48
3.7 Conclusions on e-SAFE ....................................... 51

CHAPTER 4 REMOTE AUTHENTICATION ............................. 60
  4.1 Introduction .................................................. 60
    4.1.1 Use of Error Correcting Codes ...................... 61
  4.2 Protocol Description ........................................ 63
    4.2.1 Intuition ............................................... 64
    4.2.2 Encoding at the user side .......................... 65
    4.2.3 Verification at the Server side .................... 66
  4.3 Experimental methodology for Pre-Deployment Tuning .... 68
  4.4 Putting the Methodology to Work .......................... 70
    4.4.1 Experimental Set-up .................................. 71
    4.4.2 Security Assertions After the Tuning ............ 72
    4.4.3 Further Fine Tuning of LAWN ..................... 73
  4.5 Evaluation of LAWN ......................................... 75
    4.5.1 Time Overhead of LAWN .............................. 75
    4.5.2 Power Consumption Analysis ......................... 76
  4.6 Related Work ................................................. 80
  4.7 Conclusions .................................................. 82

CHAPTER 5 CONCLUSIONS ........................................ 88

CHAPTER 6 FUTURE WORK ........................................ 89

REFERENCES ...................................................... 93
LIST OF TABLES

Table 1  Read and Write Latencies under different schemes ................. 23
Table 2  Read and Write Bandwidth under different schemes ................. 23
Table 3  Summary of the parameters involved in the design of e-SAFE ........ 33
Table 4  Experimental knobs for tuning the system .......................... 73
Table 5  Summary of the Experimental Methodology .......................... 75
Table 6  Parameters relevant for power-analysis ............................. 77
Table 7  Variables used in the analysis of Theorem 1 ......................... 91
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Shannon's Observation on Information Rate over a Noisy Channel</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Abstract view of a Distributed Secure Storage</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3</td>
<td>An Encoded Data block distributed to ( n ) Servers</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Bunching up ( V ) blocks to extract bit columns</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Layered view of Functionalities</td>
<td>19</td>
</tr>
<tr>
<td>Figure 6</td>
<td>( 10.9513,6=1212 ) Load balancing vs. Distribution: S1 and S2 are of servers. When most servers in S1 go down, file 1 becomes completely unavailable. However, if both files are split across S1 and S2, absence of no single set can make the files unavailable.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 7</td>
<td>( 10.9513,6=1212 ) Space blow-up of a document fragmented over units as a function of fraction of faulty servers. The Y-axis is plotted on a logarithmic scale. Space blow-up for an optimal error correcting code is, for tolerating up to possible malicious alterations.</td>
<td>28</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Overview of ( e )-SAFE</td>
<td>30</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Stretch Factor as a function of number of initial blocks ( (k) )</td>
<td>33</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Broad Architecture of ( e )-SAFE</td>
<td>35</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Parallel processing of File-Segment in ( e )-SAFE</td>
<td>36</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Preparing blocks for a File Segment ( e )-SAFE</td>
<td>37</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Structure of Inode for a multi-segment file in ( e )-SAFE</td>
<td>38</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Construction of Verification Information from Merkle Tree: Each server is located at a leaf node. Consider server ( S_i ), ( T_0, T_1 \ldots ) represent the nodes on the path from the root to the ( i )-th server. At any internal node of the tree the hash value ( V ) is prepared as ( V(j) = Hash(V(left_child).V(right_child)) ). The final verification information sent to ( S_i ) is the collection of hash values stored at the siblings of the vertices on this path.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Write Latency of ( e )-SAFE over Raw DHASH delivery</td>
<td>52</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Read latency of ( e )-SAFE over Raw DHASH delivery</td>
<td>53</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Split of time in Computation and Networking (Write)</td>
<td>54</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Split of time in Computation and Networking (Read)</td>
<td>55</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Write latencies of DHT and SBS implementations of ( e )-SAFE</td>
<td>56</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Read latencies of DHT and SBS implementations of ( e )-SAFE</td>
<td>57</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Write latency distribution for workload</td>
<td>58</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Back-logs during writes generated by the workload</td>
<td>58</td>
</tr>
</tbody>
</table>
Figure 23  Effect of stretch-factors on the read latency ........................................... 59
Figure 24  Property that the encoding should follow: Before encoding, \( d(A, B) \leq d(A, C) \). After the encoding the inequality should be preserved .......... 64
Figure 25  Summary of encoding and verification procedures. The user prepares \( \tau \) sketches of the same token. The verifier accepts if at least \( \tau_{\text{majority}} \) of them result in acceptable \( k \) value. ................................................. 65
Figure 26  Details of the sketch preparation .................................................................. 66
Figure 27  Comparison of Hash vectors ....................................................................... 66
Figure 28  Visualizing false positives and negatives in the correlation matrix and as function of \( k_r \) ................................................................. 69
Figure 29  Variation of expectation and the variance of observed value of \( k \) with the Hamming distance. For these experiments the string length was chosen as \( n = 4096 \) and \( \beta = 1.3 \) .......................................................... 83
Figure 30  Determining \( k_e \) and results after repeated trials, \( n, \beta \) same as before . . 84
Figure 31  Variation of \( h_1 \) and \( h_2 \) as a function of \( \beta \) keeping other parameters constant, \( n = 8192 \) ................................................................. 85
Figure 32  Results on an iPAQ .................................................................................. 86
Figure 33  Power consumption profiling for LAWN and brute force authentication (sending complete token) ....................................................... 87
SUMMARY

This thesis investigates the role of error-correcting codes in Distributed and Pervasive Computing. The main results are at the intersection of Security and Fault Tolerance for these environments. There are two primary areas that are explored in this thesis.

(i) We have investigated protocols for large scale fault tolerant secure distributed storage. The two main concerns here are security and redundancy. In one arm of this research we have developed SAFE, a distributed storage system based on a new protocol that offers a two-in-one solution to fault-tolerance and confidentiality. This protocol is based on cryptographic properties of error correction codes. In another arm, we have developed eSAFE, another prototype distributed persistent storage; eSAFE facilitates seamless hardware extension of storage units, high resilience to loads and provides high availability. The main ingredient in its design is a modern class of erasure codes known as the Fountain Codes. One problem in such large storage is the heavy overhead of the associated meta-data needed for checking data integrity. eSAFE deploys a clever integrity check mechanism using a data structure known as the Merkle Tree to address this issue.

(ii) We also investigated the design of a new remote authentication protocol. Applications over long range wireless would benefit quite a bit from this design. We designed and implemented LAWN, a lightweight remote authentication protocol for wireless networks that uses a randomized approximation scheme based on Error correcting codes. We have evaluated in detail the performance of LAWN; while it adds very low computational overhead, the savings in bandwidth and power are quite dramatic.
CHAPTER 1

INTRODUCTION: AN OVERVIEW

We envision a pervasive computing environment that consists of a continuum of computation and communication. The devices range from tiny information capture units such as cameras and microphones, huge data storage units, and many mobile wireless components all engaged with one another through the lives of one or multiple applications. Pervasive computing is not the only scenario that involves a multitude of machines. Recent systems paradigms such as Peer-to-peer networks, internet and grid computing also engender similar distributed but networked systems. For a systems researcher, there are a number of issues that come to the forefront: storage, routing, look-up, mobility, heterogeneity, data security, fault tolerance, power minimization to a name a few. In this thesis we zoom into a cross section of failure handling and security. Two primary areas that we have investigated are: (1) Fault tolerant storage protocols for large scale distributed storage and (2) A new remote authentication protocol for mobile and wireless devices. Once common theme that binds the two seemingly different areas is the use of Error-Correcting codes. In the following we briefly introduce these problems and then outline the the structure of the rest of the thesis.

1.1 Problem Statement

1. In large scale distributed data stores, the storage protocols usually suffer from different problems. Some protocols lead to serious blow up in space requirements that can result from either purely replicating the data multiple times or splitting it into too many fragments followed by subsequent fingerprinting. Some protocols, because of their dependence on symmetric key encryption, entail key-management overhead, and more seriously, are vulnerable to security threats arising out of unprotected nature of the participating clients. This thesis investigates some of the techniques to circumvent such problems.
2. Modern remote authentication systems rely on sensor-sampled data, such as fingerprints, image of a face, retinal scan and so on. In the domain of pervasive computing, where the participating entities are small wireless devices, such large authentication tokens usually consume some of the most critical resources, viz., bandwidth and power. We propose a scheme that leads to a low-bandwidth, low-power remote authentication protocol.

1.2 Large scale Distributed Storage

Pervasive systems will often generate high volumes of data through different devices, from video and audio feeds, temperature sensors, weather sensing equipments etc. Thus there is a continual need for storage. With rapidly falling price of storage units, and increasingly available bandwidth, large scale distributed storage systems are getting popular everyday. The primary benefit is high availability of service. Having a multitude of servers help in redundancy, so that in the face of failure of one unit, information can still be retrieved/processed from other non-failing ones. Decentralization can be achieved in many ways. At the simplest form, data can be replicated and each replica can be stored on an independent server. However, the space blow-up due to redundancy is very high. One can resort to the theory of error correction codes at this point, viewing the failure-prone storage units as a noisy channel and the writer of the data as a sender and the reader as a receiver. Several interesting solutions have been proposed based on this approach that usually offer highly space optimal schemes. In this thesis we describe two error correction based approaches we have experimented with, viz. an erasure code based technique and a protocol based on cryptographic properties offered by certain linear codes such as generalized Reed-Solomon codes and Goppa codes.

1.3 Remote Authentication

Error correction codes essentially add redundancy to the information. This principle can be utilized in a nice way for remote authentication purposes. Consider a mobile device that tries to authenticate itself to a server using a large authentication key (ranging from
a few kilobytes to a few hundred kilobytes), such as a finger-print. The server has in
store a template finger print $s$, to which it compares the transmitted token $s'$ and accepts
the authentication if $s$ and $s'$ are close enough. Imagine the server asking the reverse
question, i.e., instead of asking if $s = s'$, it asks if $s \neq s'$. Error correction codes can be
used nicely to verify answers to such questions. Since an error correction code adds lot
of redundancy, if the two strings $s$ and $s'$ are very different, by suitably encoding their
difference can be so amplified that it may become possible to distinguish the two even
by just randomly sampling from the encoded versions. Although such a sampling can
provide only probabilistic guarantees, instead of deterministic ones, it is acceptable as long
as there aren’t enough false positives and false negatives. Such a randomized scheme,
however, introduces a great benefit for wireless mobile devices for which power is very
critical. Typically one byte of long range wireless transmission is equivalent of millions of
processor instructions in terms of power consumption. Hence, sending a very small random
sample from an encoded signature would typically result in significantly reduced power
consumption compared to transmitting the whole key. In our work, we adapt a randomized
hamming distance computing algorithm for fast evaluation of hamming distances; we present
the theory and the experiments that validate the theory. Although, this technique is specific
to hamming distance computing at this moment, there are two reasons that underlies its
importance. First, hamming distance is a generic distance that is at the basis of computing
many other complicated metrics. Second, these techniques that are very simple in nature,
can be suitably adapted for other metrics as well.
CHAPTER 2

A SINGLE ARCHITECTURE FOR FAULT TOLERANCE
AND SECURITY

2.1 Introduction

In this chapter we introduce the problem of designing a large scale distributed storage that is secure (from the point of view of confidentiality) and fault tolerant. The importance of such a storage system is wide, especially for two reasons: first, the continuing trend of corporations outsourcing the data-storage and maintenance to third parties such as data centers, and second, the increasing level of malicious activities to threaten both confidentiality and integrity of data in such a storage.

All previously known design approaches suffer from several disadvantages, (1) \textit{High key management overhead:} For large distributed storage, key-management is a significant problem, which is further aggravated by symmetric-key encryption; using symmetric key requires maintaining a key for every writer-reader pair, which leads to potentially $O(n^2)$ keys to be maintained. (2) \textit{Insecurity due to volatile parties:} Since every participant is not equally secure, compromising the weaker parties leads to secret key being revealed. (3) \textit{Vulnerability of symmetric keys:} Since encryption and decryption are both extremely fast in this case, it is possible for an adversary to copy all the ciphertext and then try breaking the system offline without any time constraint. Sometimes even brute-force methods such as dictionary attacks can crack the encryption. Cracking password files are very common example of such attacks.

We have come up with a new design protocol to circumvent above problems. The key idea in our protocol is to use the error correcting codes as a vehicle for implementing a Public-key infrastructure. For the rest of the introduction, we first describe the problem domain in detail and also how our solution stands with respect to other design approaches.
The growth in the volume of information handled by modern applications, the falling price of storage units, and the rapid improvement in network speed 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 [20] 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. A seminal work in this area by Rabin [75] 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., when all of them are allowed to collude to extract the document. Separate 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 codes have been proposed by some researchers [4, 89]. ECC based techniques provide redundancy in a space optimal way, leading to a space-optimal design for reliability.

In this chapter, 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 linear error correcting codes (such as generalized Reed-Solomon codes and Goppa codes [74]), that allow us to use a single transformation that adds both redundancy and encryption to the data.

We have already articulated the problems of using symmetric key in the design architecture. Public-private key pairs offer better solution for all the aforementioned problems. In this case, only $O(n)$ keys need to be maintained, one pair for each participant. It reduces the threat due to volatile parties because once the data has been received by a party, and if no other copy exists, it is impossible to decipher the data without the private key. It is typically the case for the emerging systems that the recipients of the large volumes of data are secure servers, while the producer of the data are often unsafe clients. So it is worth
taking precaution against such volatile parties. Finally, public key encryption/decryption is computationally quite expensive, so it is almost impossible to deploy schemes of the flavor of offline dictionary attacks. We use this to our advantage by observing that the codes that are used to provide fault tolerance can also be exploited to provide public key infrastructure with only very small 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 standard 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 [44]. 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 error correcting 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 [37]. As we will see in later sections, some alternative design principles depend critically on hashing while our design does not.

The contribution of this work is two fold. First, we make the interesting observation that linear error-correcting codes can be used in distributed storage jointly for fault-tolerance
and cryptographic purposes. To this end, we present the design for a Secure And Fault tolerant Environment for data storage (SAFE). Next we evaluate the performance of our prototype system with respect to its most likely alternative SecureIDA. Our results reveal that we get a highly secure system, free from all the vulnerabilities of symmetric key, at a very low performance overhead.

The rest of the chapter is organized as follows. Section 2.2 presents related work, and lays the groundwork for our proposal. Section 2.3 delves into the preliminaries of error correction codes and why this is relevant to fault tolerant storage. It also talks about cryptographic properties of linear codes. Section 2.4 discusses the design of SAFE. Section 2.5 presents the evaluation of our system: the methodology and the performance with respect to its alternative. Section 2.6 presents the conclusions and the future work we intend to pursue.

2.2 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 [6] have been used to provide coordination in distributed systems. Quorum approach is 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 systems considered how to handle benign failures [36, 86]. Byzantine failures, where the servers maliciously corrupt data, and collude among themselves, were studied later on [57, 61, 58]. The replication techniques studied in these investigations were adopted in the design of persistent object stores, such as Phalanx [59] and Fleet [60].

Another alternative to handle byzantine faults in a distributed environment is replicated state machine approach [78]. Castro and Liskov [19] presented a practical implementation based on this approach; they built a file system that handles byzantine faults. The key idea 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 [62]. 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 [20] system. Rabin presented an efficient Information Dispersal Algorithm (IDA) that can be used for fault tolerance in parallel and distributed systems [75]. 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 [48] 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 [79] in a clever way that uses symmetric key encryption; the resulting scheme is shown to be secure with short secret sizes [49]. This approach, known asSecureIDA was exploited in the design of e-Vault, an electronic storage system developed at IBM [40]. A hybrid approach that combines secret sharing and replication based strategies has recently been developed by Lakshmanan et al [52]. This scheme tries to retain the good aspects of both the schemes and offers various levels of security guarantees, 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 [92]. 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 architecture [50]. Farsite [3] 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.

2.3 Error Correction and Fault Tolerance

2.3.1 Preliminaries

\[
\begin{align*}
\text{Encode:} & \quad \begin{cases} m \to n \to \text{Noisy Channel} \\ \text{(Error probability = p)} \end{cases} \\
\text{Decode:} & \quad \begin{cases} n \to m \end{cases}
\end{align*}
\]

\[
(n/m) = \text{Space blow up due to redundancy} \\
(m/n) = \text{Information Rate} \\
m/n \leq C \text{ (Channel capacity)}
\]

**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 the 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 [80]. 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 standard texts [23, 74].
Linear Codes

Next, we present a very brief exposition to linear codes. The main intent is to set the stage for introducing cryptographic properties of these codes that make them attractive for designing storage systems for untrusted environments. 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. If errors occur in not more than \( d \) positions, it is possible to detect the corruption and if the number of errors is less than \( \frac{d-1}{2} \), then it is even possible to correct those errors and recover the original string. This last phase is called decoding.

2.3.2 Cryptographic Properties of Linear Codes

McEliece Cryptosystem

Decoding a general binary linear code is NP-complete [11]. Based on a similar intuition McEliece developed a public key cryptosystem that exploits the hardness of syndrome decoding of linear codes. Although McEliece used a specific code called Goppa code, in principle any linear code fits the purpose. In brief, a Public-key infrastructure can be constructed in the following way. Suppose there is a code with the generator matrix \( G \). A public key \( G' \) is computed as \( G' = S.G.P \), where \( S \) is a scrambling matrix and \( P \) is a permutation matrix. \( G, S \) and \( P \) constitute the private key.
To encrypt a message $x$, the sender grabs the public key $G'$ and creates the cyphertext as $y = x.G' + e$, where $e$ is a random error vector having weight no more than the error correcting capacity of the code in use. Hence, the encoding is quite straightforward.

The receiver of $y$, i.e., the owner of the private key, computes the following. $y.P^{-1} = (x.G' + e).P^{-1} = x.S.G + e.P^{-1}$. Now, let $e' = e.P^{-1}$ and $x' = x.S$. Then the receiver decodes the string $x'$ by using the standard decoding procedure for the code in question (as if the message $x'$ got corrupted by the error pattern $e'$). With the knowledge of $S$, $x$ can be immediately recovered. An adversary, who knows neither $P$, nor $S$, will be unable to extract the message from $y$. The problem of finding $x$ is provably hard.

The hardness of recovering a codeword forms the basis of McEliece Cryptosystem [64]. However, linear code based public key systems never became popular as encryption schemes for a set of reasons: (1) the large size of the public key, (2) both encryption and decryption are computationally inefficient compared to other cryptosystems such as RSA or Diffie-Hellman. However, for our purpose, i.e., in the design of distributed storage systems, an error correcting code is a natural candidate for providing fault tolerance; we observe that modifying this infrastructure slightly in the line of McEliece cryptosystem, offers strong Public-key encryption at almost no additional performance cost. At the same time, all the vulnerabilities due to symmetric keys are eliminated.

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.

2.4 Design of SAFE

Figure 2 shows the block diagram of a distributed data store. There is an un-trusted storage medium. A producer($W$) 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 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 [41], 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 one of 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, nor the reader needs to maintain pairwise symmetric keys. Most importantly, the security risk is minimized; although we assume both the writer and the 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 that had access to a symmetric key, would automatically lead to the violation of confidentiality. On the other hand, it is relatively safe to assume that the consumer of the data stays

---

$^1$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.
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.

One simple alternative to the above architecture is the following. Instead of using a linear code based public-key infrastructure, a standard PKI, such as RSA is used. Now, a producer, while writing data on to the storage, generates a random symmetric key, encrypts the data using such random symmetric key, encrypts the symmetric key using the the receiver’s public key and appends the encrypted symmetric key in the file-header. Once the write is done, the symmetric key is thrown away. The receiver has to first decrypt this symmetric key and then decrypt rest of the document. However, this scheme suffers from the following disadvantage. The producers in our applications are unsafe computing agents in the sense that they run the risk of being compromised. To generate symmetric keys, they need to have a pseudorandom number generator (PRNG) running. Once compromised, the state of the PRNG would be exposed to the adversary and it may be possible to extrapolate back the numbers that have been already generated. A possible way out may be resorting to a third party for blocks of random numbers, which may not be possible in every scenario. Similarly seeding the PRNG frequently doesn’t reduce the threat either; first, a random enough source is again necessary for seeding, which gets us back to the same problem, and second, even if seeding is done some way, the sequence of numbers between two seeds, may be worked out once the internal state of the PRNG is exposed.

In our design, 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: An Encoded Data block distributed to n Servers

Figure 3 is showing the details of 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 an $m/8$ byte long data block. Once these $m$ bits are encoded to a string of $n$ bits using an ECC, the bits are distributed and are stored on different servers.

Binary versus non-binary codes: If we consider a binary code, then we have to consider the block as a string of bits (and not bytes). 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. The bit level operation can also be averted if we used non-binary codes that operate with a larger alphabet of symbols; such codes can consider a single byte as one symbol. For a non-binary code at a byte level, we need not worry about individual bits and operate at the byte-level. SAFE can handle both binary (code words are treated as bit-strings) and non-binary (byte strings) codes. Binary Goppa codes are example of the first class, while Reed-Solomon
codes are example of non-binary codes.

![Diagram](image)

**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 an 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. For a non-binary code at a byte level, \( V \) does not need to be a multiple of 8. The scheme therefore definitely yields 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. Such information is 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 the 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. Since the ECC naturally handles any alteration (less than or equal to \( f \)) in that data that may happen for any reason, which 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 it is impossible to attain 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) \([19]\). Any such mechanism can be integrated into our architecture without any incompatibility factor and incurring only performance cost.

A user, while writing in this setting follows what we already described in section 2.3. Suppose, the minimum distance of the code is \( d \). Then the decoding algorithm can correct up to \( t = \lfloor (d - 1)/2 \rfloor \) errors. The user encodes the data block using an ECC and randomly adds \( e \) bit errors \( (e \leq f) \) to the block. An adversary trying to read this encrypted data, will have to solve the Syndrome Decoding problem, which is provably hard. While, this takes care of confidentiality, the system is still robust to tolerate up to \( t - e \) failures (because the user has already used up \( e \) errors to confuse the adversary).

**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.

2.5 Evaluation

2.5.1 Objectives

![Layered view of Functionalities](image)

Figure 5: Layered view of Functionalities

In this section we evaluate SAFE on simple microbenchmarks. We also compare its performance with that of SecureIDA. To this end we used the same infrastructure that we built for developing 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. In fact, the design of eVault, the SecureIDA based system from IBM [40] has a similar architecture. The immediate purpose of our study is not the design and implementation of a file system end to end. We want to assess the practicability of our scheme outside the shell of asymptotic order complexities. We also want 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 [20], or a bunch of virtual
disks like Petal [53], 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 [85], 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 our approach 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 two 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.

2.5.2 Comparison Platform

In a SecureIDA setting, we start by encrypting a data block with some symmetric key (in this implementation we used AES in particular). We do not need to do such encryption for SAFE. For both SAFE and SecureIDA scheme, the next step is common i.e., inflate a $k$ byte data into $n$ bytes using a $(n, k, d)$ linear code. Let’s call these pieces $d_i$ for ($1 \leq i \leq n$).
For SAFE encoding, we add a random error pattern (introduce the problem of syndrome decoding for the adversary). For SecureIDA we append the vector \( (H_1, H_2, \ldots, H_i \ldots H_n) \), with each data piece \( d_i \), \( H_i \) being the hash digest of \( d_i \). (iii) Store each of these document pieces with a different server, \( n \) servers in all.

For the read operation, the decoding part is common. However, there are two small differences that translates into the performance difference.

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 increase in space.

2.5.3 Results Summary

In this section we compare the performances between SAFE and SecureIDA. As already pointed out, their mechanisms are almost identical. While SecureIDA is based on symmetric key encryption, SAFE provides the security of a public key infrastructure which is harder to crack from a hacker’s perspective. However, there are a couple of trade-offs involved. First, we sacrifice slightly in space efficiency compared to SecureIDA and second, we pay a very small performance penalty during the decoding part, i.e., in the read phase. In the following we elaborate on these trade-offs.

We assumed a total \( (n) \) of 128 servers; the maximum number of faulty servers is assumed to be \( f = 32 \). For this particular implementation, we used a Reed-Solomon Encoder/Decoder. To sustain maximum of 32 faults, the SecureIDA scheme takes 64 bytes and inflate them into 128 bytes. Finally, a bunch of such blocks are assembled into a matrix and respective columns are isolated and sent to respective servers. Space blow up for this scheme is therefore 2. We disregard the size of the fingerprints which in practice adds up slightly to the space. We used a similar Reed-Solomon encoder for encoding in the SAFE case; however the encoded had to be protected by other operations that we have already discussed. While encoding in the SAFE scheme, it is necessary that the writer introduces a few random errors. We assumed a maximum of 10 such errors. Therefore, to correct against
the combination of $f = 32$ server failures and $e \leq 10$ randomly introduced errors, we need a minimum distance of $2(e + f) = 84$ in our code. Thus, keeping the maximum length of the codeword at 128, we can accommodate only 44 message bits; i.e., we convert every block of $k = 44$ bytes into a block of $n = 128$ bytes. The space blow up therefore is $n/k \approx 3$.

Table 1 presents the write and read latencies under different schemes. The leftmost column indicates the number of blocks chunked together before extracting the byte columns. For SecureIDA, these blocks are 64 bytes. For SAFE, the blocks are 44 bytes long. As we can see, the time taken by SAFE for write operations is slightly but consistently lower than SecureIDA. Two factors account for this difference. First, unlike SIDA, no additional symmetric key encryption is required for SAFE. Second, the encoding takes slightly lower time to inflate 44 bytes into 128 bytes than performing the same operation on 64 byte blocks. Introducing additional errors in the case of SAFE takes negligible time.

The read operation, however, is complicated. In this case, SAFE performs slightly worse than SecureIDA. This is because, apart from the data communication, while SecureIDA performs decoding and decryption, SAFE has to perform decoding followed by an unscrambling operation. Although the decoding algorithm takes almost the same time for both, the unscrambling is computationally more expensive than decryption with symmetric key, which accounts for the slight degradation in the latency. Similarly, Table 2 illustrates the bandwidth consumed by the two operations in two different cases. The interesting observation here is that although the latency for writes in SAFE is slightly smaller, the effective bandwidth is actually lower. By bandwidth, we mean the exact amount of the original data is written or read per unit time. For each write, in SAFE a block of 128 bytes contain 44 bytes of the original document and 84 bytes of redundancy information, while for SecureIDA the same block contains 64 bytes of original information; hence the difference in effective bandwidth.
Table 1: Read and Write Latencies under different schemes

<table>
<thead>
<tr>
<th>#Blocks</th>
<th>Write (sec)</th>
<th>Read (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIDA</td>
<td>SAFE</td>
</tr>
<tr>
<td>512</td>
<td>0.18</td>
<td>0.163</td>
</tr>
<tr>
<td>1024</td>
<td>0.32</td>
<td>0.29</td>
</tr>
<tr>
<td>1536</td>
<td>0.46</td>
<td>0.41</td>
</tr>
<tr>
<td>2048</td>
<td>0.61</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Table 2: Read and Write Bandwidth under different schemes

<table>
<thead>
<tr>
<th>#Blocks</th>
<th>Write (KB/s)</th>
<th>Read (KB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIDA</td>
<td>SAFE</td>
</tr>
<tr>
<td>512</td>
<td>182</td>
<td>138</td>
</tr>
<tr>
<td>1024</td>
<td>205</td>
<td>155</td>
</tr>
<tr>
<td>1536</td>
<td>213</td>
<td>164</td>
</tr>
<tr>
<td>2048</td>
<td>215</td>
<td>163</td>
</tr>
</tbody>
</table>

2.6 Discussions

We discussed a new design principle, i.e., that of using error correcting code based approach for building secure, fault tolerant storage. The design of SAFE exploits this principle. There has been a concern regarding the kind of code to be used for the encryption. The principal criterion to be fulfilled is the following: for an adversary, who just knows the scrambled and permuted generating matrix, there are too many possible codes to guess [5]. In his original proposal, McEliece suggested using Goppa codes; there are an abundance of Goppa codes for a given parameter set so that the adversary has no idea which one to follow for decoding purposes. Niederreiter proposed a scheme based on generalized Reed-Solomon codes. Gibson [34] constructed a structural attack on this scheme. There are similar attacks known on other kind of codes as well. For a vivid exposition, the reader can consult [34]. However, although the structural attacks indicate that such systems can be broken, it is not completely true; in reality, they only prohibit using codes of small parameters such as length and dimensions. For standard cryptographic applications such parameters are critical and need be small, otherwise the key sizes become significantly large. For our purpose, i.e., very large scale distributed storage systems that seem to be the emerging storage technology,
the limitation on key-size does not apply, because error correction is already mandatory for redundancy reasons. Also, the structural attacks become successful if the number of possible codes are very few. It is always assumed for Reed-Solomon codes that for an underlying field of order \( q \) the length of the code is \( n = q - 1 \). This assumption is made because \( n = q - 1 \) yields maximum information rate. However, it need not be true in general and if \( q \) is much larger compared to \( n \), it opens up combinatorially explosive number of codes, (although not the highest information rate for a given field size). However, for our purpose, it is quite useful to exploit this property.
CHAPTER 3

AN EXTENSIBLE STORAGE

3.1 Introduction

One major problem in designing large scale distributed storage systems is to have the ability to expand the storage in a seamless fashion. In this chapter we articulate the benefits of extensibility and then seek to address the challenges that come in the design path of an extensible storage system. Our main motivation is the storage technology's perceivable shift from centrally managed data servers to distributed units. Consider an organization such as a corporate house or a university that has a large set of machines; the collected capacity of their local disk-spaces far exceeds those of the typical file servers dedicated to host data for them. Moreover, it has become evident, that for storage, cost of management will strictly dominate that of the hardware. Thus the new paradigm is geared toward reducing cost of ownership of data [21]. While the immediate benefit is robustness, an economic impact is reduced management overhead; since information would be efficiently replicated and scattered into many packets, one need not worry about a few (or possibly many) packets that may be lost due to hardware or software malfunctions.

In this chapter, we describe e-SAFE, a robust, storage system that is tailored to the requirements of this upcoming paradigm. In the process of developing e-SAFE, we borrowed from a spectrum of design principles, both theoretical and engineering, and deployed them coherently into a single system architecture. We focus on a very large scale storage system that is quite common in large organizations. Such hardware infrastructures typically grow very fast in size, are bound to work under very high loads and are required to provide high availability to the data that is stored within. Consequently our design goals differ substantially from a peer-to-peer file sharing system. However, our design principles borrow from the insights gained through research in the space of distributed and P2P computing.
Enabling Technologies

For quite a few years now, researchers have been investigating the design of massively large scale storage systems that can potentially span the internet. Paradigms such as Peer-to-Peer systems and the Grid computing are bringing diverse computing domains within cooperative environments in order to harness and utilize computing resources more effectively. While traditional P2P systems have been explored to carve out storage-utilities [84, 50, 77], significant research has been done in enabling storage-area-networks (SAN) that can truly span large geographical distributions [90, 91]. Thus traditional storage systems have been evolving to become internet-wide with IP based underlying communication subsystems (IPSAN [73]).

3.1.1 Motivation and Challenges

![Diagram of Traditional Load Balancing vs Redundancy + Load Distribution]

**Figure 6:** Load balancing vs. Distribution: S1 and S2 are set of servers. When most servers in S1 go down, file 1 becomes completely unavailable. However, if both files are split across S1 and S2, absence of no single set can make the files unavailable.

3.1.1.1 Load, Availability and Distribution

Large scale data servers typically serve hundreds of clients simultaneously; thus, although such systems enjoy abundance of disk-space, the units have to serve under very heavy workloads at recurring intervals, which calls for revisiting availability of data under high load conditions. The higher the load on a server, the longer is the response time. For a client
issuing a read request therefore, data becomes practically unavailable if it is not retrieved within a threshold latency period. Thus, a high workload condition, which is often the case in data intensive application domains, resembles a low availability scenario. Standard load balancing strategies, when applied to data storage, would skew the data distribution in an unfavorable way. Figure 6 shows how load balancing can skew the data distribution. The main idea is that if the files are always assigned localized storage units, the failure of one unit will always result in the unavailability of some file. On the contrary, stretching the files across servers seems to be a better alternative. If the degree of distribution is high with a high stretch factor, a client can read data with a reasonably low latency even in the face of a very high workload that renders most of the servers unavailable. In section 3.2, we will discuss quantitatively the effect of redundancy on load balancing.

3.1.1.2 Key Issues

As systems scale up, component failures become more of a norm than an isolated event. Thus scaling up demands robustness. Our specific goals for the design of a robust storage system include reducing management overhead that manifests itself in two forms: (i) Repair overhead. In the face of failures a monitoring unit has to keep track of any data loss and follow up with appropriate recovery as well as create regular back-ups, and (ii) Handling Extensibility. Because of the rapidly falling price of disks, it is easy to conceive that storage system of an organization will see rapid growth with time, enabling further reduction in cost of ownership for individual storage units. So, from a management point of view, while it is lucrative to further reduce the cost of ownership by creating additional redundancy over the extension units, it is also essential that such readjustment be seamless and fast. Traditionally, replication or erasure codes are used for these purposes. While the former is extremely space inefficient, the latter, i.e., the traditional erasure codes are limited in a number of ways to incorporate scaling up; explicit parameter tuning is necessary. Added to that are algorithmic performance penalties.

---

1 stretch factor s is defined as the factor of redundancy, i.e., a file of size f, after adding redundancy bytes s.f bytes long.
3.1.1.3 Hazards of Fragmentation

![Fraction of Faulty Servers vs. Space Blow-up Factor](image)

**Figure 7:** Space blow-up of a document fragmented over \( n \) units as a function of fraction of faulty servers. The Y-axis is plotted on a logarithmic scale. Space blow-up for an optimal error correcting code is \( \frac{n}{n^{2t+1}} \), for tolerating up to \( t \) possible malicious alterations.

It is well understood that fragmenting data over a multitude of servers provides higher availability [89, 12, 14]. Under a given error rate, it can be shown that the lifetime of data (the time after which retrieval becomes practically impossible due to errors creeping in the storage systems) increases rapidly with the number of encoded fragments. However, modern systems are constantly beset with security threats from agents that not only cause fail-stop behavior, but can maliciously alter the information, which may go undetected unless otherwise safeguarded. Corruption of documents can also be the result of software faults; for example when the hardware or operating systems are upgraded in massively distributed systems, small patches of invisible incompatibility factors (such as a deprecated driver) may introduce wrong reads or writes. Such experiences have been reported in recent literature such as the Google File System [35].

A standard way to safeguard against such malicious failures is to attach a fingerprint vector [75, 49, 48]. A fingerprint of a large document is typically its cryptographic hash, i.e., the digest obtained by a one-way hash function so that the probability that two randomly
chosen documents will produce the same hash value is negligibly small. Alon et al. observed how such schemes can result into prohibitive space blow up under a very high failure rate [4]. Intuitively, as the fragmentation increases, for a given file, the fragments become shorter while the fingerprint vector grows larger, and thus the verification information starts dominating over the actual information. Figure 7 shows how the blow-up factor grows at a high fault rate. The space-blow-up factor is plotted on the Y-axis in a logarithmic scale. As the number of fragments $n$ increases, the blow-up increases drastically; similarly the blow-up sharply jumps as the failure rate increases for any given $n$. For example, if a file of size 100 KB is split across 100 servers designed to tolerate $t = 50$ faults using some standard optimal erasure handling techniques such as Reed-Solomon codes, one can derive that for every 2 KB of original information, there will be 2 KB of additional verification information [4]. These numbers get unfavorably biased towards this overhead as $n$ grows larger. It may be argued that this blow-up is only limited to the fingerprints, which is independent of the file size. However, it is often not possible to process an entire file in memory. Thus a file needs to be treated in multiple segments large enough to fit in the main memory resulting in similar overhead for every segment of the file.

### 3.1.2 $\epsilon$SAFE

The designer of a large scale storage system thus confronts the following tension: On the one hand, the continuous growth of hardware infrastructures, the understanding of the probability of availability and the possibility of parallelizing data processing and dissemination, all hint at fragmenting data over as many servers. On the other hand, the associated performance cost of reaching out for too many pieces and the resulting growth in verification information hint at limiting the number of encoded pieces. $\epsilon$SAFE is designed to strike a balance; while $\epsilon$SAFE can efficiently take advantage of as many storage units as possible in a seamless manner, at the same time, the overhead of verification information is limited to a logarithmic blow-up factor (by novel use of a data-structure known as the Merkle-Tree).

The design of $\epsilon$SAFE rests upon a few key principles: (i) Use of a specific class of erasure codes called the Fountain Codes for seamless extensibility and fast encoding/decoding, and
(ii) Efficiently replicating (by Fountain codes) and sprinkling data all over, so that high availability and load resilience can be guaranteed. To support the design, e-SAFE also has the following optimization gears that enhance its performance: (i) Task parallelization over multiple file segments that can take advantage of a parallel processing hardware such as a symmetric multi-processor (SMP), (ii) A background dissemination mechanism, that exploits lazy intervals between I/O bursts to disburse replicated information, hiding the cost of replication and dissemination from the user.

The rest of the chapter is organized as follows. The next section presents an overview of e-SAFE and the rationale behind its design. In section 3.3 we describe the main architectural components of e-SAFE followed by a short discussion of its implementation in section 3.4. Section 3.5 presents the evaluation of e-SAFE . We discuss the related work in section 3.6 and finally conclude in section 3.7.

3.2 System Overview

3.2.1 Design Overview

![Figure 8: Overview of e-SAFE](image)

To the user e-SAFE offers a file system interface just like NFS. Underneath is a distributed storage system. Figure 8 depicts the broad design of e-SAFE . A small set of machines (denoted as the Directory Server) serve as the meta-data server for the documents. We assume this set is highly secure and reliable and data stored in here is modifiable only
by authorized access. Since this is a very small set, we believe these assumptions are not very restrictive from security management point of view. Underneath the directory server, a host of machines constitute the distributed block store. These machines can be distributed, from the span of a single building to the scale of geographically scattered locations. They are potentially fault prone. While locating a file, a file-system user locates the path of the file from the directory server, which provides with the file-metadata (viz., inode) that contains information about the distribution of the actual data blocks over the block-store.

Documents are encoded using a class of rate-less erasure codes (called the Fountain codes). The output of the encoding, a sequence of small blocks, is sprinkled across multiple storage units. We discuss in section 3.2.2 why it is important for us to have a large and flexible stretch factor. The idea is that even if some of the pieces are corrupted, there is no need to spend any maintenance effort for recovery purposes.

Documents stored in e-SAFE are not immutable, however, it is optimized for a class of access that mostly appends blocks to files. It is possible to edit a document from the middle as well. We assume that the meta-data information for every document (such as an inode) is stored in a machine that is highly secure, i.e., unauthorized modifications are not possible. We kept document encryption and key management outside e-SAFE. However, any traditional encryption mechanism can be integrated within the structure of e-SAFE without any compatibility issue. While we decided to keep the confidentiality issue outside our design goal, checks for data integrity are quite stringent.

3.2.2 Design Rationale

We investigate the question: What does availability mean in the context of a distributed set of servers? In a P2P setting, availability may simply mean the percentage of time a server is up. However, in a non-transient context, this definition is not appropriate. A good way to express the availability of a server is through the load. The availability index can be expressed as:

\[
\text{Availability Index} = \frac{\text{real response time}}{\text{optimal response time}}
\]

Once the above index is less than a threshold \( \text{Av}_{\text{min}} \) the system may be considered
unavailable from a client’s point view. Thus the availability \( \mu \) can also be expressed as the probability:

\[
\mu = \text{Prob}(\text{Availability Index} < \text{AV}_{\text{min}})
\]

\( \varepsilon \)-SAFE is targeted for very large files for which redundancy using replication becomes very expensive. As already discussed in section 3.1, we used Fountain codes. The unique property that such codes offer is a complete flexibility of how much a file can be stretched. Suppose a file segment is coded in the \( n : k \) ratio, i.e., \( k \) blocks are encoded into \( n \) blocks, then, ideally one could reconstruct the original segment from any \( k \) blocks. Thus, given an average availability \( \mu \) for each block, the net probability that the segment can be retrieved is equal to the probability that any \( k \) or more blocks be available. Thus the expression is straightforward:

\[
P(a|v) = \sum_{j=k+1}^{n} \binom{n}{j} \mu^j (1-\mu)^{(n-j)}
\]

Simplifying the above equation, Bhagwan et al. [13] derived a direct expression for the stretch factor \( c = \frac{n}{k} \),

\[
c = \left( \frac{\sqrt{\frac{\mu(1-\mu)}{k}} + \sqrt{\frac{\mu^2(1-\mu)}{k} + 4\mu}}{2\mu} \right)^2
\]

Figure 9 shows how the required stretch factor varies as a function of \( k \), the number of pre-encoding blocks, keeping the availability \( \mu \) constant. We show two cases for \( \mu = 0.1 \) and \( \mu = 0.2 \). It turns out that there is a sharp decrease in required \( c \), as \( k \) goes beyond a particular threshold. And after the sharp fall, the curves flatten out, as if increasing pre-encoding fragments does not entail further stretching. In this work, we consider high load situations; we are not particularly interested in high host probability. Thus we consider cases with \( \mu \leq 0.1 \), which means that on the average each server (or storage unit) is so loaded as to respond at most one out of 10 times, within the acceptable latency period (specified by \( \text{AV}_{\text{min}} \)).

So we have two points to explore in the design space. First, use a high \( k \) and use a smaller stretch factor. The advantage is space-efficiency, while the drawback is high fragmentation
Figure 9: Stretch Factor as a function of number of initial blocks \(k\)

Table 3: Summary of the parameters involved in the design of \(e\)-SAFE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_s)</td>
<td>Length of a segment</td>
</tr>
<tr>
<td>(n_b)</td>
<td>Number of blocks after encoding</td>
</tr>
<tr>
<td>(L_{meta_data})</td>
<td>Length of a meta-data block</td>
</tr>
<tr>
<td>(c)</td>
<td>Stretch factor</td>
</tr>
<tr>
<td>(f_{over_head})</td>
<td>Maximum allowed overhead fraction</td>
</tr>
<tr>
<td>(N_{max}, N_{min})</td>
<td>Upper and lower bounds on (n_b)</td>
</tr>
<tr>
<td>(AV_{min})</td>
<td>Minimum availability threshold</td>
</tr>
</tbody>
</table>

with additional meta data overhead. Second, use a smaller number of pre-encoding blocks \(k\) and stretch further so as to cover low-availability cases. This approach is less space-efficient, but has the following advantages: \(i\) Since during a read operation data must be constructed from at least \(k\) fragments, it is usually more efficient to keep this number low and \(ii\) System can handle write operations in a more time-distributed fashion; first it writes a small number of blocks just enough to reconstruct data, and then when the write burst gets over, lazily disseminate other redundancy blocks onto the persistent storage. Thus the user programs do not see high write-latency.

Keeping in mind the two factors, we designed \(e\)-SAFE to automatically decide the most suitable strategy within a set of constraints. Three key variables play critical roles in
the choice of parameters: Length of a segment \( (L_s) \), the number of post-encoding blocks \( (n_b) \) and the size of meta-data \( (L_{\text{meta data}}) \) that needs to be appended to each data block. e-SAFE tries to find the best parameters within a given set of constraints, such as the maximum number of servers allowed, maximum meta-data overhead allowed and so on. For a single segment that is disseminated with a stretch factor \( c \) (resulting size \( c.L_s \)) and over \( n_b \) blocks, e-SAFE chooses the variable parameters so the meta-data overhead is less than a given threshold \( f_{\text{overhead}} \):

\[
\frac{n_b \cdot L_{\text{meta data}}}{c \cdot L_s} \leq f_{\text{overhead}}
\]

subject to the constraints: (i) \( N_{\text{max}} \geq n_b \geq N_{\text{min}} \) and (ii) \( c \leq c_{\text{max}} \). Table 3 summarizes the meaning of all the parameters used in the discussion above.
3.3 Architecture

![Diagram of e-SAFE architecture]

**Figure 10:** Broad Architecture of e-SAFE

Figure 10 gives a high level view of e-SAFE architecture. The top layer is a *file-system client* that provides read, write interfaces to the user. The next layer, FTM (Fault Tolerance Module) gathers/sends data stream to and from the FS-Client. FTM performs erasure encoding and decoding operation on the data streams. The key idea that FTM uses is that of a *Fountain Code*. We describe it shortly in section 3.3.1. The output of FTM is a series of data blocks. The next stage, *viz.*, the Verification layer, prepares fingerprint information for the data blocks that it receives as input. The fingerprint is generated by one way cryptographic hash function such as SHA-1. To survive malicious faults, *i.e.*, alteration of data, or forging of identity of storage units, fingerprint of individual blocks are also then linked together into a data structure called the Merkle Tree. This is described in section 3.3.2. Finally, once the data blocks are appended with appropriate fingerprint information, they are handed down to the data dissemination/aggregation layer. This stage of the software decides where to locate the server (or the storage unit) for storing or retrieving data. Presently, the search is done by computing the hash of the block-content and then indexing into a distributed hash-table. Once this decision is taken, a request
followed by the payload is sent to the respective File-system servers. The FS server is a simpler structure. On the one hand it implements an RPC-based messaging protocol with the dissemination layer in the client, On the other hand this layer maintains a database of blocks indexed by their hash-values. Thus it can store or retrieve a block for the client as requested.

![Diagram](image)

**Figure 11:** Parallel processing of File-Segment in e-SAFE

Figure 11 shows the optimization gear inherent in e-SAFE architecture. A large file is divided into multiple segments. Each segment is passed asynchronously to the FS-client. FS-client in turn calls the subsequent modules. The process is parallelized over different segments. Concurrent invocation of modules overlaps computation/communication of multiple blocks.

Figure 12 shows the flow of events in processing a file segment. The file segment is divided into $K$ blocks and then expanded into $N$ blocks using erasure codes. The ratio $N/K$ is the stretch factor of this code. Usually it is more efficient (and guarantees higher accuracy for probabilistic codes) to choose large $K$ and $N$. Out of the $N$ small fragments, we coalesce $N/n$ of fragments together to obtain $n$ new blocks that are independently handed down to the block-stores underneath. Note that coalescing does not change the fault tolerance limits of the system, i.e., if the encoding can tolerate $f . N$ failures out of
\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{figure12.png}
  \caption{Preparing blocks for a File Segment e-SAFE}
\end{figure}

$N \ (f < 1)$, then after coalescing, at most $f.n$ can get faulty out of $n$ blocks. The scaling property inherent in the erasure codes helps us adjust the scale of the system quite easily.

To support such parallel dissemination of segments, we have to maintain separate metadata structures for each segment. Figure 13 shows how this is organized. Each file has one master Inode. The master maintains pointers to the indirect Inodes corresponding to each file segment. A segment Inode contains the content-address (typically the hash) of the various blocks within this segment, and other information such as number of blocks, the encoding parameters and so on. Retrieval of Data blocks is simple - the content hash is obtained from the Inode and then the main block is restored from the block-store underneath. However, when the block-store underneath is not content-addressable, an explicit mapping needs to be maintained.

When the number of blocks ($n$) is large, it becomes inefficient to keep the user waiting in a blocking mode until all the blocks are stored. Thus the control returns to the user only after a subset of blocks are stored (a subset just large enough to reconstruct the segment). The write occurs in burst. Thus, the client maintains a queue of blocks in the machine's `\texttt{\textbackslash temp}` directory and in the gap between write operations, unobtrusively and asynchronously disseminates rest of the blocks. This way one can provision for very high stretch factors.
and thus for very high load factors.

*e*-SAFE provides two modes of file store privilege to the user: *Permanent store with versions* and *modifiable*. For the former, files are never removed, rather each version is maintained, with a version number. Typically the most recent version is returned, however, any previous version can also be retrieved. Maintaining consistency is trivial. In the latter case, a client, with appropriate permission, can modify existing files. In this case, the modification is done at the segment level. First, the directory server maintains consistency by serializing all write operations on a file, *i.e.*, when a file is being modified by one user, no other user can modify it. A file modification essentially means generating new segments and deleting the old ones. The inodes at the directory servers are updated with the pointers for the new segments and similarly the segment inodes with the address of the new blocks. For the old blocks, requests are sent to the appropriate nodes, so that they can reclaim the disk-space occupied by the orphaned blocks. However, complete reclamation of these disk blocks cannot be guaranteed by *e*-SAFE, because we don’t assume the nodes to be non-malicious all the time, rather accept that faults (including malicious ones) are a fact of life.

Figure 13: Structure of Inode for a multi-segment file in *e*-SAFE
3.3.1 Erasure Coding

An erasure code works in the following way: A document is partitioned into \( k \) blocks. These are called the \textit{message blocks}. Next \( n \) new blocks \((n > k)\) are generated from them by adding some redundancy mechanism, such as addition of extra parity bits and so on. The new blocks are called \textit{encoded blocks}. Later on, the original document can be constructed from any \( k \) out of \( n \) encoded blocks. There are many ways erasure codes can be generated. A very standard way is the use of the Reed-Solomon codes [74]. However, these codes are inflexible in the sense that the parameters \( n \) and \( k \) are static and cannot be changed on the fly. The decoding time for such codes are \( O(n^2) \), which means for large \( n \) they become impractically slow. Moreover, these codes operate over mathematical structures called the finite fields. Once a field size \((q)\) is chosen, it is not possible to change it later. The maximum length of one symbol that can be considered as one unit of encoding is limited to \( \log q \), placing a limit on the width of the stripes necessary to distribute data blocks amongst multiple storage units.

To overcome all of the above problems we used a modern class of erasure code called the \textit{Fountain Codes} [18]. The specific version that we used is known as the LT (Luby Transform) codes [55]. LT codes are rate-less, in the sense that the stretch factor \( n/k \) can be varied on-the-fly. Plus there is no limit on the symbol size - thus no restriction is imposed on the striping. And finally, these codes are inexpensive to implement and very fast in encoding (linear) and decoding times \((O(n \log n))\). The category-name \textit{fountain} is suggestive - when one needs to fill a cup of water from a fountain, it does not matter which particular droplets are being collected; rather just enough number of drops needed to fill the cup is sufficient. Rate-less codes can produce a \textit{fountain} of encoded blocks from \( k \) original message blocks. For a pre-decided small number \( \epsilon \), only \((1 + \epsilon)k\) number of data blocks out of this fountain will suffice to reconstruct the original document. Our idea is to collect the blocks and sprinkle them over to as many storage units as necessary and thereby secure a very high \textit{durability guarantee}. 

39
Figure 14: Construction of Verification Information from Merkle Tree: Each server is located at a leaf node. Consider server $S_i$, $T_0, T_1$... represent the nodes on the path from the root to the $i$-th server. At any internal node of the tree the hash value $V$ is prepared as $V(j) = Hash(V(left\_child), V(right\_child))$. The final verification information sent to $S_i$ is the collection of hash values stored at the siblings of the vertices on this path.

3.3.2 Verification

We mentioned in the introduction that as data gets fragmented over more and more servers there is an impractical space overhead incurred by the verification information; especially in the face of high failure rate. To overcome this problem we took help of a data structure called the Merkle Tree (MT)[65]. An MT is simply a tree (assume binary tree for the time being) where each node $j$ contains a hash value $V(j)$. The value is obtained from the child nodes; suppose $L_j$ and $R_j$ are the left and right children of the node $j$, then $V(j) = H(L_j, R_j)$, where $H()$ stands for a one-way cryptographic hash such as SHA-1 and the dot denotes concatenation operation.

We use the MT structure in our system in the following way. Suppose a file (or one segment of a file) is being disseminated amongst $n$ storage units. Let’s assume that $n = 2^l$ for some integer $l$. Now consider an MT of depth $\log n$ and $n$ leaf nodes such that each server can be mapped to one leaf node. The hash value associated with a leaf node is the hash of the data block being sent to the corresponding server. Once the leaf hash-values
are directly obtained from the corresponding data blocks, the intermediate levels of hashes are easily computed all the way up to the root. Figure 14 depicts this MT. Now, consider the $i$-th server (alternatively, the $i$-th leaf node), and the unique path from the root to this node. The vertices on this path are $T_0, T_1$ and so on. We collect the hash values associated with the vertices that are siblings of $T_1 \ldots T_5$. Since $T_0$ has no sibling, we keep the hash value at $T_0$ separately as the Root-Hash. In the figure, all the siblings are marked by a 'square'; their values are collected and packed into the verification information sent to the $i$-th server. While retrieving a block, one needs to re-compute the hash and then successively recompute the hashes up its path to the root (with the help of the sibling hash-values stored) and then finally verify if it is matching with the root-hash. If this test is passed, the data block's integrity can be considered intact with very high probability.

3.4 Implementation

c-SAFE has been implemented as a user level library written in C++ and has been tested to run on Linux. We used SHA-1 for hashing purposes, for verifying data blocks as well as creating block identifiers. There are two implementation of the data dissemination layer. The first one uses DHash [25], a Distributed Hash Table (DHT) based peer to peer system, built on top of a scalable look-up system called Chord [84]. In our DHash based implementation of c-SAFE, each inode maintains a list of its block identifiers. During a store or fetch operation, lookup is initiated into the Chord network. Once a machine with appropriate nodeID is obtained, the data block is directly exchanged with it. The Inode server does not need to exclusively maintain the mapping between the blocks of a file and the servers they are stored in; that is left to DHash. Although Chord was designed to cater to P2P like systems, it has been later used to build wider area file system. Moreover, DHash performance has been enhanced (by a factor of two) by the use of a new transport scheme called the Striped Transport Protocol(STP). Thus we chose to use the STP based DHash implementation for our purpose.

We also implemented another version of the block store that does not depend on DHash. In this implementation, the Inode server maintains the mapping between blocks and the
servers they are stored in. We will refer to this implementation as the Simple Block Store (SBS). Currently, the blocks are randomly assigned to servers from a list of server hosts, and this information has to be explicitly maintained at the Inode server. This way of choosing servers runs the risk of skewed load distribution (which is one of the reasons why we first chose DHash as our implementation vehicle). For SBS, once the client obtains location information for a block, it contacts the respective servers through an RPC interface. The end-to-end performance of this implementation is better than DHash, because the chord-based lookups are replaced by a direct client-server contact. SBS has two versions, synchronous and asynchronous. The former is based completely upon the synchronous RPC implementation available in Linux. The latter is an extension to it. In the asynchronous version, the RPC client maintains an extra thread receiving any incoming packets from the server asynchronously. Similarly, on the server side, an asynchronous service thread is maintained, along with a request queue. The standard RPC request handler enqueues requests that are asynchronously serviced by the service thread.

The synchronous SBS is simpler in its design and easy to implement and its mode of operation is RPC-based. It is not necessary to have an extra service thread and thus no context switch is required. On the contrary, the asynchronous version needs an extra thread and additional context switches and thereby incurring slight inefficiency when the servers are lightly loaded. However, in a high workload situation where many of the servers respond with high latency, the asynchronous version is the right choice because it enables a client to contact the servers in a non-blocking mode (in contrast, a synchronous client would block on the server response).

3.5 Evaluation

We evaluated e-SAFE both on a series of micro benchmarks as well as with workloads. The objective of the microbenchmarking was two fold: (i) quantify at a micro level how much the read and write operations cost over the implementations of the block store, and (ii) break down the total times taken to inspect how different layers contribute to the total cost of operations.
All our experiments were performed on a set of fourteen dual processor SMP machines, 2.8 GHz Intel Xeon processor, with 2 GB SDRAM and 512 KB L2 cache. They are connected by a switched Gigabit Ethernet. In all the following experiments, the machines are partitioned into two physical sets - servers, hosting the block-store (acting as e-SAFE FS-server in Figure 10), and the clients that are outside the ring of servers, and only generate store and fetch operation for the servers. The client machines do not store anything. This division was effected partly because we wanted to separate the performance interferences of clients and servers and also to mimic the structure of a separate storage subsystem catering to its clients.

3.5.1 Microbenchmarks

We primarily study the latency of e-SAFE for storing and fetching files of different sizes. The latency increases as we split files across more and more servers, i.e., split files into more and more blocks. e-SAFE is limited in performance by the bandwidth offered by the block-store underneath. Thus we examined e-SAFE under two different implementations of the block-store that we described in section 3.4.

Performance of e-SAFE on DHash

End-to-end Latency

The first experiment that we performed is the following. An e-SAFE client writes a file of size \( s \) onto the store, split into \( b \) blocks. The latency is recorded. To compare with the above operation, the same file, split into equal number of blocks, is handed over to the raw DHash layer for storage. Since e-SAFE performs additional operations, such as stretching the files by encoding operation, processing and adding verification information, and appending metadata that is needed to support subsequent decode and verify operations, the expectation is that e-SAFE would perform worse compared to the raw DHash layer. Figure 15 shows the comparisons for different file sizes. The X-axis represents the number of blocks (essentially the number of different servers the file was distributed to) and the Y-axis shows the observed latency. We show the comparative latencies for various file sizes. From the latency values we see that e-SAFE follows quite closely the performance of DHash. For example, for a
file size of 10MB, and around 300 blocks, the overhead of e-SAFE is around 50\%(15000 sec vs. 10000 sec). However, this difference remains almost constant across all levels of fragmentation. These numbers show that the top three layers of e-SAFE (FS-client, FTM, and verification) together do not add significantly to the overall latency.

Figure 16 describes the results of similar experiments for the read operations. In this case, e-SAFE latency follows the baseline DHash latency, however, as the data size increases, we see that this gap widens. e-SAFE still performs quite well though; for example splitting a 10MB file onto approximately 300 blocks, yields a latency of less than 3 seconds and thus yielding a throughput of over 3MB/s.

Writes are more expensive than reads. During a write operation, the scheme produces many more blocks than the original document has based on the chosen stretch factor. In the above experiments, we used a stretch factor of 2.0. However, higher latencies are expected (and observed) as we increase this factor. Reads however are somewhat independent of the stretch factor. A read succeeds as soon as enough number of blocks are gathered. Plus, caching in the DHash layer facilitates the reads by reducing the look-ups. Note that the latency numbers are expected to be poor compared to standard network file server (NFS) throughput. However, NFS operations involve single RPC call between a client and the server. NFS does not provide any kind of fault tolerance either. In contrast, e-SAFE has to make RPC calls to a multitude of servers, and thereby securing a high degree of fault tolerance at the cost of disk space and networking time.

**Dissection of Cost**

Figure 17 gives a breakdown of the latency (write operation) into two parts: the time spent in the top three layers (labeled encoding), and time spent in the dissemination layer (labeled networking). The numbers clearly indicate that additional computations done by e-SAFE are quite minimal compared to the time spent in dissemination.

Figure 18 breaks down the time for read operations. Here we see that the computation time dominates the network time. There are two reasons for this result. First, as we already pointed out, it is not necessary to collect all the data blocks that were originally written,
a reduction in network time compared to writes. In addition, the computational overhead for a read is much more than a write for the following reason. Although the verification process (during read) is slightly faster than generating verification information (during write), the high decoding cost during the read offsets the verification thus inflating the absolute cost of computation compared to the writes.

**SBS Performance**

We have performed similar experiments on the second implementation of the data dissemination, *viz.*, the SBS implementation. Since this implementation explicitly maintains a mapping between block ids and servers, no look-up is necessary. Figure 19 presents the numbers for the write operations (in a similar experimental setting as before). As before, the X-axis denotes the number of blocks and the Y-axis denotes the latency. We observe that SBS writes are much faster compared to DHT based e-SAFE. As we already pointed out, SBS does not need any lookup to find an appropriate server, since the block-to-server mapping is readily available in the Inode server. We see this across all the data sizes (only a small representative set is presented here). The performance of the SBS version is much better than the DHT based implementation of e-SAFE; with reference to figure 20(d), a file of size of 10MB is written in roughly 2 seconds, yielding a bandwidth of 5MB/s (block-size = 1024) compared to the few hundred kilobytes of bandwidth obtained using the DHT implementation.

Figure 20 shows the performance of read operation. Here we see extremely fast reads happening in this setting. While the performance cost of the DHT-based implementation keeps increasing with the number of blocks, the general trend of the SBS graph is horizontal. For example, for a data size of 10KB, and split amongst 128 blocks, the DHT implementation takes around 550 ms, while the the latency of the SBS implementation is only 100 ms. Such a result is in accordance with the expectation. While true P2P systems will benefit from the DHT based systems, our results indicate that for applications running over systems of the size of not more than few hundred machines, SBS implementation is more suitable.
3.5.2 Performance under Workload

Next we tested e-SAFE under a synthetic workload since we do not have real I/O traces. Our purpose is to generate synthetic workloads that can generically mimic the characteristics of many different realistic workloads.

I/O workload characterization has received wide attention in the past, and experiences reveal that typical read/write requests that occur over various domains and scales (such as disks, network, web) follow some well defined patterns [39, 54, 24, 88]. First, I/O is bursty - both read and write requests appear in short bursts with intermediate lean periods. Second, I/O traffic bears strong self-similarity, i.e., if one zooms into smaller and smaller intervals of an extended I/O trace, the patterns over smaller intervals resemble those over longer intervals. Third, there is a strong resemblance to these traces with the 80-20 rule often observed in Database systems [42]. This rule roughly says approximately 80% of the query traffic experienced by a server actually queries for 20% of data stored in it. To model such behaviors, we used a trace generation model called the bmodel [88]. It has been shown that this model accurately approximates I/O behaviors of various systems, especially disk I/O. The model is dependent on a parameter b, called the bias. A bias of 0.8 corresponds to the factor 80% in the 80/20 law. bmodel also generates self-similar traffic. Self similarity in traces is usually measured by an index H, known as the Hurst Exponent [39]. The Hurst exponent of a trace generated by a bias b is given by: $H \approx \frac{1}{2} - \frac{1}{2}(b^2 + (1 - b)^2)$. Thus using bmodel allowed us to vary the characteristic of the traces to create a family of work-loads that are fairly generic and representative of the I/O encountered by storage systems.

We have carried out the following experiments with the workloads generated by bmodel. Synthetic I/O traces are generated for both reads and writes for a given time interval. Next we initiate a client to write/read by following those traces. Figure 21 presents a scenario of this workload for write operation. A total of 100 MB of data was distributed to be written over a period of 200 seconds. The distribution (into different chunk sizes as produced by the bmodel), is spaced evenly over the 200 sec interval. In Figure 21, X-axis denotes the workload index (i.e., the indices of write requests of various data sizes), and on the Y axis, we measure the latency each write request takes to finish. Since the
requests arrive in that order in time, the X-axis can be also be treated as a virtual time axis (for arrival of write requests). \textit{bmodel} generates approximately 130 different requests of various sizes within this interval (X-axis is marked from 0 to 140). We record latencies for two different modes of storing; background dissemination on and off. The lower line (write completion with background transfer) represents the former, while the upper line represents the latter. The shape of the curves show the bursty nature of the generated workload with the peaks corresponding to writes of larger blocks. Clearly, the user experiences a lower latency with the background transfer on, because much of the data dissemination happens in the background after the write-call returns. The stretch factor in these experiments is kept at 2.0. As the I/O burst continues, the write requests queue up building a backlog. The system cleans up this backlog during the lean phases of I/O, that come immediately following the bursts.

In Figure 22, we show how the backlog builds up. Again, The X-axis denotes the requests arriving in that order. The Y axis represents the elapsed time. The lowest line represents the precise arrival time of requests relative to the workload window. The middle line represents the completion time for the corresponding write operations with background transfer enabled, i.e., storing just enough for reconstruction, and leaving the rest for the background dissemination. The top line represents the completion time for the corresponding write operations when the data dissemination happens entirely in foreground. Clearly, the background process does reduce the backlog, however, it does introduce partial backlogs, i.e., parts of incomplete writes. For reads that happen long time after the writes, the background dissemination would clear up this partial backlog. The large sizes of the local disks help us contain these backlogs. The lazy dissemination has another benefit. For reads that are too closely spaced with the write, the data blocks just written may still be readable on the local disk, thus avoiding network penalty for such reads.

Finally, we wish to demonstrate the effect of higher stretch factors that we outlined in section 3.2. For this experiment, we use the asynchronous version of the SBS. For the experiments reported here, we use a file of size 240 KB. We store this file using stretch factor 2 and 4. In the former case, we split the file into 16 blocks, and in the latter 64
blocks of the same size as in the former case. Now we perform fetch operation on the file assuming the servers are highly loaded. We simulate a loaded server in the following way; for returning every block a random delay is inserted between 0 and δ time units. A higher value of δ would emulate higher load on the servers. The two results are plotted in Figure 23. δ was varied from 10ms to 10 seconds. As we have reasoned before, the response for the case of stretch factor 4 is much better than the case of stretch factor 2; it is literally ten to hundred times better. Such a result indicates the great benefit that a flexible stretching scheme can provide, viz., the ability to attain high availability in a decentralized storage where most of the units run under heavy load. This result also underscores the importance of using Fountain codes that offer the flexibility of dynamically increasing the stretch factor of files as the system size scales up.

3.6 Related Work

Distributed Storage is an age-old concept that has received wide attention in multiple contexts. Thus, a long list of related literature predates our work.

Distributed Storage Systems

Distributed storage hardware such as RAID [20] continue to be improved in the current practice. Systems such as Petal [53] seamlessly exports through a distributed set of disks to clients. For large scale organizations, having a multitude of storage units, the natural evolution was to have systems such as a Storage Area Network (SAN) or network attached storage (NAS) [68]. Modern systems are emerging on the Internet scale such as the Internet Protocol Storage Area Network (IPSAN) [73]. Clearly, such growth in scale, and the fall in price of storage hardware provides the infrastructure for systems such as e-SAFE to be realized and operational; this way our work is complementary to the growth in the area of large scale storage devices.

Close to our work is the area of distributed file systems. A few recent examples are xFS [7], Frangipani (on top of Petal) [85], and so on. With the growth of the Internet, the focus is shifting to realizing file systems over wider areas. Security is one of the most critical issues. SFS is a file system based on separation of key management from file system
so that it can span the Internet with heterogeneous key management policies [63]. The key idea in SFS is the use of self-certifying path name. FARSITE system [3] enables a peer to peer like environment of mutually distrusting desktops to provide a highly available and secure file system; it takes care of making copies of replicas and metadata when some of the nodes decide not to participate in the storage anymore. Security involves two aspects, viz., confidentiality and integrity. While, confidentiality is not a big problem in systems that are centrally maintained (regardless of being distributed), data integrity is critical, because faults, both in the form of fail-stop and data corruption creeps in for various reasons. Wide area file systems, such as CFS [26], make sure that integrity and availability are maintained with effective replication caching. The Google file system (GFS) deploys a distributed file system spanning literally thousands of servers that deliver high bandwidth and availability [35]. GFS provides availability by splitting files into blocks and then replicating each block along with additional checksums, very much along the line of CFS.

Unlike SFS or FarSite, for e-SAFE confidentiality is kept out of its design. However, there are similarities in the way all these systems manage their metadata. e-SAFE focuses on data integrity in a space optimal way. e-SAFE is similar to GFS and CFS in the sense that they all fragment files and store them onto multiple units. The SBS implementation of e-SAFE functions like the master server (a dedicated server holding all the mapping between blocks and destination servers) based implementation of GFS. The DHT based implementation functions very much like the CFS. However, e-SAFE differs from both CFS and GFS, FARSITE and other similar file systems in the way redundancy is added. While others rely on pure replication, e-SAFE uses Fountain erasure codes as its redundancy mechanism. Erasure codes are known to provide more space optimal schemes than pure replication and hence e-SAFE provides higher storage optimality for a similar level of fault-tolerance (compared to CFS and GFS).
**P2P File systems and look-ups**

In recent years research in peer-to-peer systems has received very wide attention. While the span for P2P systems is the Internet, the scale is also overwhelmingly large, i.e., literally millions of servers and their dynamics need to be considered. P2P systems open up a plethora of new possibilities, of which file sharing became popular even at a commercial level. Oceanstore [50, 76] system first attempted to harness the astronomically high amount of storage that might be reached through the Internet to create an archival level persistent storage that provides very high availability and durability. One of the key ingredients in its design is the use of erasure codes. Erasure codes have been shown to provide higher durability guarantees than naive replication at a much lower space cost [89]. Very recently, Bhagwan et al. reported TotalRecall (TR), a file system that is designed to handle the dynamic behavior of P2P systems. TR is based on the observation that in a P2P system, nodes join and disappear in a diurnal pattern over a short run, and in the long run many of the nodes leave the system permanently. To handle such scale of dynamics, TR deploys an availability monitoring unit, that checks for the availability of files on a periodic basis and repairs a file back to the desired availability whenever this factor falls below a threshold. Understanding the dynamics of P2P systems has been an important ingredient of the understanding of the availability [12, 13, 15]. In has been sometimes argued that in a P2P like environment, only a very small fraction of the nodes cooperate meaningfully and on a permanent basis, while the rest disappear mostly after a short period. However, we do not assume an environment such as P2P. We target a domain that resembles a server farm that is built out of many inexpensive local disks available individually. In our design we have decided to use a high stretch factor to replace periodic repairs. However, simple stretching will not work in a P2P context, because irrespective of the stretch factor, data, will be permanently lost in the face of constant decay [13]. For a storage system that is built out of a non-transient server-farm (e.g., a corporate environment, or grid based data storage facility), constant decay of data is not a concern. As we have discussed earlier, the presence of high workloads may create the illusion of unavailability of the data. Prior Studies and
experiences with the P2P systems have inspired some of our the design decisions. For example, a high workload condition in e-SAFE is similar to short term unavailability in a P2P setting; this similarity help us derive the conditions for the required redundancy factors.

3.7 Conclusions on e-SAFE

In this chapter, we have discussed the design of e-SAFE, a distributed storage service targeted for very large scale decentralized systems. In the design of e-SAFE we have made a quantitative observation, that of equating high load with low availability. At the heart of e-SAFE 's design, is a special class of code, called the Fountain codes, that makes e-SAFE seamlessly adaptable to unlimited stretching and thus to hardware extensions of any degree. By virtue of the same codes, e-SAFE can sprinkle data around to any stretch factor, and thus can reduce management overhead to a great extent.
Figure 15: Write Latency of e-SAFE over Raw DHash delivery
Figure 16: Read latency of e-SAFE over Raw DHash delivery
Figure 17: Split of time in Computation and Networking (Write)
Figure 18: Split of time in Computation and Networking (Read)
Figure 19: Write latencies of DHT and SBS implementations of e-SAFE
Figure 20: Read latencies of DHT and SBS implementations of e-SAFE
Figure 21: Write latency distribution for workload

Figure 22: Back-logs during writes generated by the workload
Comparing Fetch Latency for various stretch factors

Figure 23: Effect of stretch-factors on the read latency
CHAPTER 4

REMOTE AUTHENTICATION

4.1 Introduction

We study the problem of remote authentication over long range wireless networks using large signature keys (e.g., biometric samples such as fingerprints, retinal scan etc.).

Given the advent of pervasive existence of connectivity and computation, the issue of remote authentication is extremely relevant to a multitude of emerging applications. One can easily envision people accessing confidential documents from secure information stores onto devices such as handhelds or cell-phones. Current technology such as Authentec’s AES2510 fingerprint sensor [1] is readily integrable with devices such as a cell phone. However, authentication protocols are needed to enable other primary transactions and should not add significant overhead in terms of power or bandwidth. Within a given energy budget therefore, one would like to authenticate as many sessions as possible. However, a practical deployment must address the following issues:

(i) Large Key Size: In a sensor-based authentication, one important and often overlooked benefit (in addition to reduced user intervention) is the large size of the signature keys (such as biometrics). Large keys are desirable for their high entropy, but at the same time would add to the transport overhead.

In addition to large key sizes, for applications involving potentially highly mobile clients that may communicate over frequently re-adjusted ad hoc network routes, the problem is aggravated for the following reasons:

(ii) Reconnection activities: The devices in the new paradigm are mostly mobile; hence the connections are often reset and reestablished. Discontinuity in connection has been studied in prior works such as [82, 44]. However, every reconnection will require fresh authentication adding a significant overhead.

(iii) Continual Nature of Authentication [46]: As the new devices become equipped with
increasing variety of sensors, it is natural to expect that confidentiality checks be performed at regular intervals without interrupting the user activities; smaller the session length, the harder it is for an adversary to break in.

Repeated authentication using a large key adds tremendous overhead to the original transactions. Thus the scheme will benefit from a mechanism that can reduce the transmission requirement yet retains the high entropy advantages of a large key. A potential solution to key length could be sending a cryptographic hash instead of the whole key. Although in some application specific cases (such as voice samples) hashing might be possible [67], finding a generic cryptographic hash could be difficult for two reasons. First, the conditions in which two samples are taken vary considerably over time; for example, fingerprints of a person taken by the same device during different working conditions will most likely turn out to be different. Second, sensors have inherent inaccuracy and sampling differences [17, 29]. Therefore, two samples of the same object, such as a fingerprint, generated by two different sensors are most likely not identical. Cryptographic hash functions however, do not usually preserve distances, and hence two samples of the same object may result in very different digests at different conditions.

4.1.1 Use of Error Correcting Codes

In this paper we discuss a technique that enables remote authentication over long range wireless in a continual fashion even with very large keys; this scheme results in significantly reduced consumption in bandwidth and hence power. Our approach is motivated by the concept of a holographic proof [71, 83]. A proof is called holographic if it is so constructed that its verifier does not need to scan through its entire length, but the verification can be done by only examining randomly selected small parts of the proof.

How does one construct such a proof? Error correction codes (ECC) provide a candidate infrastructure. Normally an error correction code expands a bit-string by adding redundancy so that even in the presence of corrupted, or missing bits, the original data can be recovered. ECC can play a role in authentication in the following way. Consider the primary task of authentication: One party presents an authentication token (s) to a server which matches
this token against a stored template ($s'$) to see if $s \approx s'$, i.e., if there is a close(or complete) match between the two. However, one could reverse the question, i.e., ask if the two strings are different (or very different). To answer this reverse question, an error correction code ($E$) can be used to transform $s$ and $s'$. Now, take a small random sample $e$ from $E(s)$ and a corresponding sample (i.e., using the same random indices) $e'$ from $E(s')$. An estimate of the distance between $s$ and $s'$ can be made from the distance between the two small samples $e$ and $e'$. $e$ and $e'$ will be called the sketches of their corresponding patterns. Typically a sketch is very small compared to the original string and consumes orders of magnitude less bandwidth and power for transmission.

**Hamming Distance based computation**

As we have already mentioned, sensor-based samples can be quite diverse, such as fingerprint, retinal scans, voice or even non-biometrics. Typically a pattern matching algorithm is used to compute the distance between patterns in the process of verification. Unfortunately, it is not obvious how to construct an encoding scheme given an arbitrary pattern recognition problem. In this paper, we therefore focus on a generic problem, that of computing the Hamming distance. The Hamming distance between two strings is simply the number of bit positions they differ in. It is one of the simplest distance metrics that lies at the heart of many pattern classification algorithms. For example, in a retinal scan match, retinal features are extracted in a binary feature vector and then two binary feature vectors are compared for their Hamming distance.

In this paper, we propose LAWN, a new light-weight protocol for remote authentication. It has the following features: (1) It is simple and computationally efficient, (2) It trades computation for communication based on the realization that CPU cycles consume much less power compared to wireless transmission; the trick lies in sending a small sketch instead of a full authentication key. Under a reasonable power model, we show that LAWN can save up to 80% of power compared to transmitting the whole key. The bandwidth gain is obvious. (3) Since a small sketch is transmitted instead of the whole key, it becomes almost impossible for an adversary to reconstruct the original key from the sketch - the benefit
comes as a by-product.

Contributions

Perhaps the most interesting contribution of this work is the observation that error-correction based randomized schemes can be used in the domain of mobile and ubiquitous computing to reduce bandwidth and power requirements. We present LAWN, a remote authentication protocol that achieves this objective. We prove the security guarantee of LAWN (Theorem 1, section 4.2.3). Moreover, there is a wide gap between theoretical prediction and a real life system; in reality, before deploying LAWN as a working system, extensive experimentation is needed. We bridge this gap by describing a systematic and structured experimental methodology to tune the parameters of LAWN; this is a very critical component toward a real-life implementation. Once the pre-deployment tuning is done, LAWN is very easy-to-implement. We evaluate LAWN’s performance by running the prototype implementation on iPAQ 3870, over wireless IEEE 802.11b and note that the timing overhead incurred by LAWN is quite acceptable given the power-saving. Finally we quantify the savings in bandwidth and power for long range wireless (cell-phones communicating over hundreds of meters) and note to our great satisfaction that the gains in both departments are quite dramatic.

The rest of the chapter is structured as follows. Section 4.2 describes the scheme in detail. In section 4.4 we carry out a complete evaluation; we demonstrate that the scheme can provide strong security assertion and how to fine tune the systems parameters to accommodate the application-specified requirements. We also present LAWN’s computational overheads and projected power consumption overhead, both by simple analytical modeling and via estimation over real platforms running working codes. Section 4.6 discusses the related work and we conclude in section 4.7. The proof of our main theorem is presented in the Appendix.

4.2 Protocol Description

In this section, we describe the process of preparing the sketch of a signature key. First, in section 4.2.1 we start by elaborating on the intuitive principle that we laid down in section
4.1.1. In two subsequent sections we describe the encoding scheme and the verification process respectively. Section 4.3 presents the experimental procedure for determining the values of the system parameters.

4.2.1 Intuition

![Diagram](image)

**Figure 24:** Property that the encoding should follow: Before encoding, \(d(A, B) \leq d(A, C)\). After the encoding the inequality should be preserved.

What property should the encoding follow so that a reasonable distance approximation can be made from a small sketch? Encoding using an error correction code usually blows up the distance between two strings. Refer to Figure 24. Suppose \(A, B, C\) are three strings of length \(m\). An error correcting code \(E\) ensures that the distance between any two inflated string is always more than a threshold value, say \(\delta_{\text{min}}\), i.e., \(d(E(A), E(B)) \geq \delta_{\text{min}}\), and similarly for other pairs. Additionally, in our case, \(E\) should also preserve the distance inequalities; if, \(d(A, B) \leq d(A, C)\) before encoding, the inequality \(d(E(A), E(B)) \leq d(E(A), E(C))\) should be preserved.

Not every ECC meets the above requirement. A simple coding scheme toward this goal is the following. Append additional redundancy bits to a string. The redundancy bits should be direct function of the bits from the original string; they could be simple parity evaluation, or some complicated hashing, or even polynomials evaluated at various points of an underlying field.
for (i = 1 to \(\tau\))
begin
  for (l = 1 to t)
  begin
    \(X_l = l\) random bits from \(X\)
    \(v^i(l) = Hash(X_l)\)
  end
  Create i-th hash vector
  \(v^i = (v^i(1) \ldots v^i(t))\)
end

Communicate the vector
\(V = [v^1, v^2 \ldots v^\tau]\)

<table>
<thead>
<tr>
<th>Encoding at the user-side</th>
<th>Verification at the remote end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive (V = [v^1, v^2 \ldots v^\tau])</td>
<td>(g = 0;)</td>
</tr>
<tr>
<td>for (i = 1 to (\tau))</td>
<td>for (i = 1 to (\tau))</td>
</tr>
</tbody>
</table>
begin | begin |
| \(Y_i = \) same bits from \(Y\) | \(Y_i = \) same bits from \(Y\) |
| \(u^i(l) = Hash(Y_i)\) | \(u^i(l) = Hash(Y_i)\) |
end | end |
Compute \(k\) from \(v^i\) and \(u^i\) | Compute \(k\) from \(v^i\) and \(u^i\) |
if \((k \geq k_r)\), then \(g++\); | if \((k \geq k_r)\), then \(g++\); |
end | else Reject |

**Figure 25**: Summary of encoding and verification procedures. The user prepares \(\tau\) sketches of the same token. The verifier accepts if at least \(\tau_{majority}\) of them result in acceptable \(k\) value.

### 4.2.2 Encoding at the user side

Figure 26(a) captures the encoding. \(M\) is the authentication token of length \(n\) that the user wants to communicate to the server. The encoding can be thought of as a standard coding scheme such as appending extra parity bits to the end of a data block. In the figure, \(x\) denotes a subset of bits taken from \(M\) (the bits in \(x\) are not necessarily consecutive in the original string \(M\)). The redundancy bits \((R)\) in the encoded string consists of hash values taken from all such strings \(x\) of all possible lengths. A simple example of hashing is taking the parity of \(x\).

How to sample from the redundancy information? We borrow the sampling technique from the work of Cormode et al. [22]. Figure 26(b) shows how a small subset is selected from the encoded string. The elements of \(R\) are the hash digests of subsets \(x\) of \(M\). Assume that they are ordered alphabetically and according to the length of \(x\). A small hash vector is constructed from \(R\). A set of \(x\)-values are picked, say \(x_1, x_2 \ldots x_i\), in the increasing order of their lengths, i.e., \(|x_1| < |x_2| < \ldots < |x_i|\). And then from \(R\), their hash values are collected into a small hash vector \([H(x_1), H(x_2) \ldots H(x_i)]\). We call this hash vector the *sketch* of \(M* \)

---

65
and for the rest of the chapter we will use both the terms interchangeably. The substrings $x_1, x_2, \ldots, x_t$ are random with the restriction that $|x_1| = \beta$, $|x_2| = \beta^2$, \ldots, $|x_t| = \beta^t$. The parameter $\beta > 1$ is determined experimentally as we shall show later on.

### 4.2.3 Verification at the Server Side

![Comparison of Hash vectors](image)

**Figure 27:** Comparison of Hash vectors

Figure 27 describes the actions taken on the server side. The main steps are summarized...
in right half of the figure 25.

We assume that the server shares the same pseudo-random generator and it has the full knowledge of the indices of the substrings \(x_1, x_2, \ldots \) etc. Server prepares a similar sketch using the same random positions. Then it compares the corresponding positions of the two sketches. Suppose that they differ for the first time in their \(k\)-th coordinate. Notice that with increasing coordinates, a hash vector contains hash values for substrings of increasing length. If the original two strings, \(i.e.,\) the user token \(M\) and the server template \(T\) differ in many positions, there is higher chance that they would differ in a smaller substring picked from the same set of locations and therefore most likely would differ in their hash values as well. Intuitively therefore, for strings having large Hamming distance, the value of \(k\), an indicator for the first length of substrings that differ in hash values, would be small. And a high value of \(k\) would indicate that the strings are perhaps not too different from each other. Thus there is a strong correlation (inverse) between \(k\) and the Hamming distance \(h\) between the strings being matched which is captured by the following theorem.

**Theorem 1** Let \(h\) denote the hamming distance between two tokens of length \(n\). There exists a small constant \(0 < \mu_0 < 1\), such that for any hash collision probability \(\mu(\mu \leq \mu_0)\), a security parameter \(0 \leq \epsilon \leq 1\), and \(\beta \geq 1\), the following inequality is satisfied with probability at least \((1 - \epsilon)\).

\[
\frac{h}{n} \leq \frac{1 + \mu}{\beta} \left(1 - \left[\frac{1 - \epsilon}{1 + \mu}\right]^\beta\right)
\]

In other words, the above theorem states that given an observed \(k\) value, one can estimate an upper bound (the right hand side of the inequality) on the Hamming distance, so that the real Hamming distance is not more than this bound with a very high probability \((1 - \epsilon)\). In order to keep the discussion simple, we defer the proof to the Appendix. However, in the rest of this subsection we want to underline the main take-away from the above relationship. To set the stage for the take-away message, we first define a very important quantity that is based on the inverse relationship between \(h\) and \(k\) presented above.

**Definition 1** Cut-off-\(K(k_r)\): Let the verifier observe a \(k\) value after comparing two sketches.
The value $k_c$ is defined so that whenever $k > k_r$, the verifier accepts the sketch and whenever $k \leq k_r$, the verifier rejects the sketch.

Need for Experiments: The existence of a cut-off $k$ is inspired by the result in Theorem 1 that provides a probabilistic guarantee that given an observed $k$, the Hamming distance is most likely upper-bounded by a quantity derivable from the values of the system parameters. However, in reality the Hamming distance could be much smaller than the upper bound. While it is wise to be conservative in estimates, there is a chance that going by just the upper bound results in paranoia, i.e., many legitimate strings get rejected. Consider the need of any application; the typical requirement is a sharp-tailed statistical relationship between $h$ and $k$ at an application-specific point, i.e., if the application wants to accept all strings with Hamming distance less than 15% (of the string-length), and reject all strings with distance more than 10%, then there should be a sharp change in the observed value of $k$ across this point. However, Theorem 1 does not sharply locate $k_r$. We need to find $k_r$ by performing statistical experiments. In the following we will describe our methodology to perform the experiments and then the results of the experiments will be discussed in section 4.4.

4.3 Experimental methodology for Pre-Deployment Tuning

The determination of a cut-off $k$ ($k_r$) has the following requirement to start with. Given a threshold Hamming distance $h_c$, all strings with $h \geq h_c$ should be rejected and the rest accepted. The choice of $k_r$ decides the correctness and sensitivity (number of misjudgement) of the decision.

We now describe how to make a choice of $k_r$ on an experimental basis, by performing the following experiment. We randomly generate a large number (a few million) of template tokens and for each of which we generate a large number of variation token, i.e., strings with various Hamming distances. For each template and a variant of known Hamming distance, we compute the sketches and obtain the $k$ values with random trials.

The above experiments help us fill what we call the correlation matrix(shown in figure 28(a)). The rows are marked by increasing Hamming distances; this axis in a sense is the
control variable for the experiment. The columns are marked by increasing $k$ values, i.e., the values of the observed variable. The $(i,j)$-th entry in this matrix denotes the probability that a string of Hamming distance $h_i$ resulted in the observed value $k = k_j$. Note that for a given row $h_c$ and column $k_r$, the bottom right rectangle of the matrix therefore embodies the total probability of false positives, i.e., strings which in spite of having a larger Hamming distance ($h \geq h_c$) give rise to $k \geq k_r$. Similarly, the upper left rectangle of the matrix signifies the total probability for false negatives, i.e., the strings having a low Hamming distance yet rejected because of lower $k$ values observed. Before deploying the system, it is reasonable to experimentally estimate these probabilities. Once this matrix is estimated, the choice is determined by the need of the application. For every combination of $(h_c,k_r)$, there is a specific ratio of false positives and false negatives; the application can choose the cut-off for which this ratio is the most appropriate. Figure 28(b) shows how the false positives and negatives typically vary with the cut-off $k$. As we already discussed, and quite intuitively, setting a high cut-off $k$ would result in a drop in false positives, but also in the rise of false negatives. We observed from our experiments that it is quite reasonable to choose the cut-off that marks the intersection of the two graphs (the false positives & the
negatives). This will be a reasonable choice provided the two curves do not intersect at a very high value of their own. In section 4.4 we will show that this is indeed the case, and that such choice for the cut-off does not make a significant compromise in security.

To boost the success probabilities further, the experiments can be repeated multiple times (say $\tau$) and then the final acceptance/rejection decision can be taken on a majority basis. The sketch is computed $\tau$ times (independently and randomly) and then the final message is a collection of all the sketches. At the verification end, $k$ is computed for each sketch and the string is accepted if $k \geq k_r$ for at least $\tau_{majority}$ times ($\tau/2 \leq \tau_{majority} < \tau$).

Through the above methodology the value of cut-off $k$ can be determined. At this point, a little more accuracy can be accomplished by tuning the system parameters such as $\beta, \tau, \tau_{majority}, k_r$ etc. We refer to this last process as the fine tuning. In the fine tuning step, the system is evaluated (for false positives and negatives) over a small parameter space around the parameter values already set, and then the most effective point in that small neighborhood is chosen as the final parameter configuration for the system.

The steps involved in getting LAWN to work are summarized here. Note that the first two steps are one time operations and need not be performed again unless application specific cut-off values or token lengths change. (i) Pre-deployment Tuning: Given an application specified cut-off Hamming Distance, experimentally find out cut-off $k$. (ii) Fine tune the system parameters to improve upon the false positives and negatives. (iii) Deploy the code of sketch computing on both ends of the verification. Accept whenever observed $k$ is more than cut-off $k$, reject otherwise.

### 4.4 Putting the Methodology to Work

Now we present real life experiences in carrying out the aforementioned methodology for pre-deployment tuning of LAWN. We describe the experimental set-up in section 4.4.1 and then in section 4.4.2 we evaluate the security in terms of false positives and show that the probability of a breach can be made negligible. Finally in section 4.4.3 we show how we fine tuned the system for more efficient results.
4.4.1 Experimental Set-up

Our first step is evaluating the correlation matrix \( C \) presented in figure 28(a). To do so, we have randomly generated many \( (10^6) \) templates. For each such template, we generated a large number \( (10^4) \) of proof strings of different Hamming distances. We varied these Hamming differences starting from 1\% up to 40\% of the length of the original string. For each proof string the hash vector is computed and the same is compared with the hash vector generated from the corresponding template string. Each comparison results into an observed value of \( k \). If the Hamming distance is \( h \), the \((h,k)\) entry in the correlation matrix is incremented by one. Finally to convert the values to probabilities, all entries are normalized between 0 and 1. After \( C \) is determined, we determine the cut-off \( k \). Note that this depends on the cut-off Hamming distance. Typically, the applications generate legitimate proof strings that always lie within a certain distance from the original. Let’s denote this distance (as a percentage of the length of the token) by \( h_1 \). This means, any string within percentage distance \( h_1 \), should be accepted with high probability. We call \( h_1 \) the \textit{Accept Threshold}. Similarly, there is another boundary, \( h_2 \geq h_1 \) such that any string with percentage distance more than \( h_2 \) should be rejected with very high probability. We call this the \textit{Reject Threshold}. While the gap between the two thresholds is application specific, a non-zero gap is quite natural. In other words, if a fingerprint that has a percentage difference of 5\% with the template should be accepted with a high probability, another string with a distance of 6\% should also be accepted with equal or slightly less probability. In summary, there should be a gradual fall in acceptance rate from \( h_1 \) to \( h_2 \); after \( h_2 \), the acceptance rate should become as close to zero as possible. We choose a cut-off Hamming distance \( h_c \), which is in the middle, \( h_1 < h_c < h_2 \). \( h_c \) could be chosen in many ways, as linear or non-linear function of \( h_1 \) and \( h_2 \). While determining systems parameters, \( h_c \) can be chosen experimentally by evaluating a large subset of values lying between \( h_1 \) and \( h_2 \). Once, the cut-off Hamming distance is fixed, the next task is to determine the cut-off \( k \) from the correlation matrix. We have already described this procedure in section 4.3.
4.4.2 Security Assertions After the Tuning

Now we describe the main results of our experiments and demonstrate the validity of our methodology. The main objective of this section is to demonstrate that despite the probabilistic nature of the scheme, it is possible to tune the system so as to yield very low rate on false-positives, i.e., in other words, allow very little security breach. The process consists of two simple steps: (1) choose a suitable cut-off \( k \) and (2) prepare multiple sketches of the same token and take the majority estimate (boosting of success probabilities by such repeated trials is well known).

We have performed our experiments with different string lengths and over varied range of other parameters. Figure 29(a) depicts the expected value of observed \( k \) as a function of the Hamming distance in a typical experiment, generated from the normalized entries of the correlation-matrix. As expected, the expected value of \( k \) decreases with increasing percentage distance. Figure 29(b) presents the variance of observed \( k \). Interestingly enough, the variance too decreases monotonically with increasing percentage difference. Such a diminishing variance helps in reducing the false positives, as we shall note later on in this section.

We have experimented with several values of \( h_1 \) and \( h_2 \). We present a typical set of results (Figure 30(a)) just as a guideline of how things turn out for the experiments. We set \( h_1 = 6\% \) and \( h_2 = 14\% \) and \( h_c = 10\% \). This choice is somewhat arbitrary. However, the overall pattern of our experiments is quite independent of the specific choice of \( h_1, h_2, h_c \) which is normally a requirement of the application. Figure 30(a) presents the false positives and negatives as a function of the cut-off \( k \) for \( h_c = 10\% \). This means, if we set \( k_r = 8 \), we will have more than 20\% false positives (unacceptable) and similar number of false negatives. Following our method, (prescribed by figure 28(b)) we set \( k_r = 9 \), that jointly minimizes the false positives and negatives. Then we repeat the experiments for \( \tau = 5 \) times. Repeating the experiment means preparing 5 different hash vectors on independent random trials. The string is accepted if \( \tau_{majority} = 3 \). Figure 30(b) depicts the results after these repeated trials; the Y-axis denotes the normalized number (of accepted and rejected strings respectively). All strings with percentage distance less than 5\%, are accepted with
high probability. Similarly, almost all the strings with distance more than 14%, are rejected.
Note that in this case the false positives consist of all the accepted strings that have Hamming
distance higher than 14%, i.e., their number is given by the the area under the curve
“accept” (figure 30(b)) from 14% and beyond, which is visibly negligible. We also note here
that once a legitimate token is rejected (false negative), the whole token can be sent again
so that the verifier can deterministically take an authentication decision.

**Asymmetry between false positives and negatives:** An interesting observation
follows from the above results. Although the rate of the false positives are quite low, the
false negatives could not be minimized equally. To understand this, we go back to figure
29(b). Note that the variation of observed $k$ decreases with increasing Hamming distance. In
other words, the observed $k$ values get increasingly localized for higher Hamming distances.
Therefore, it becomes easier to identify such strings with greater confidence than the ones
having low Hamming distances. Because of the large variance of $k$, strings with lower
Hamming distance, are likely to generate $k$ values much lower than the expected ones.
Such strings are rejected, and thereby constitute the false negatives; these are represented
by the area under the “reject”-curve in figure 30(b) between zero and 5%. This asymmetry
between false positives and negatives is also a testimony to the theoretical result by Pang
and Gamal [70] that a two way approximation bound on the Hamming Distance is not
possible with logarithmic number of bits.

### 4.4.3 Further Fine Tuning of LAWN

<table>
<thead>
<tr>
<th>$h_c$</th>
<th>In the range $[h_1 \ldots h_2]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$, $\tau_{majority}$</td>
<td>change $h_1$, $h_2$ and their gap</td>
</tr>
<tr>
<td>$\theta$</td>
<td>For adjusting $h_c$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>control $h_1$, $h_2$, false positives and negatives</td>
</tr>
<tr>
<td>$\mu$ (hash collision probability)</td>
<td>very little effect</td>
</tr>
</tbody>
</table>

We now show how to fine tune LAWN. We start with the application specific accept
and reject thresholds, viz., $h_1$ and $h_2$. (As we have already seen we start by choosing a
cut-off distance $h_c$ midway between $h_1$ and $h_2$.) As before, the application would specify the Accept and Reject thresholds ($h_1, h_2$). Setting $h_c$ arbitrarily however, yields two different threshold values, ($\hat{h}_1$) and ($\hat{h}_2$). The objective of the experimental tune up is to find out the settings so that $\hat{h}_1 \approx h_1$ and $\hat{h}_2 \approx h_2$. We stress the fact that the following is a fine-tuning, not a trial-error over a big range of parameters. Typically, for most of our experiments, choosing $h_c = \frac{h_1 + h_2}{2}$ yields reasonably good results, given a $\beta$ so that $\beta^t$ covers a large fraction of the token length ($n$).\(^1\) However, slight adjustments of the parameters yield further improvements.

To give further control over the system, we introduce one more parameter $\theta$. Once we determine $k_r$, we can set our cut-off k-value to $k_c = k_r + \theta$, so that we accept when $k \geq k_c$ and reject otherwise. This new parameter $\theta$ is an integer and could be positive or negative depending on whether we are willing to admit a few more false negatives or positives respectively. Table 4 shows the parameters that can be controlled during the phase of experimental tuning.

Figure 31 shows some details on fine tuning experiments. To emulate an application we assume two completely arbitrary values for accept and reject thresholds, say $h_1 = 5\%$ and $h_2 = 17\%$. Now, we try out different cut-off Hamming distances, i.e., we vary $h_c$ in the range 5\% to 17\%. In Figure 31 we show some of the choices. Figure 31(a) shows results with $h_c = 12\%$, with $\tau = 5$ repetitions and $\tau_{majority} = 3$ for acceptance. In this experiment we vary $\beta$ (represented by X-axis) over the range 1.45 - 1.57 with increment of 0.1. On the Y-axis we measure the $\hat{h}_1$ and $\hat{h}_2$ respectively. Figure 31(b) represents similar experiments with $\tau = 7$ and $\tau_{majority} = 4$, while Figure 31(c) represents experiments with $\tau = 7$ and $\tau_{majority} = 4$ and $h_c = 11\%$. In 31(a), the range $1.5 \leq \beta \leq 1.55$ yields the required cut-off values. This range is highlighted with an oval on the X-axis. In 31(b) we obtain a relatively smaller set of values producing the same effect; again, these are marked with an oval. Figure 31(c) yields similar results. We finally choose one of the configurations from 31(a) as we again experimentally determine that these yield the best values for false positives and negatives.

\(^1\)t = maximum sample length, ref. section 4.2.2, for all our experiments we kept $t \leq 30$
Table 5: Summary of the Experimental Methodology

<table>
<thead>
<tr>
<th>Input Specification for LAWN</th>
<th>Experimentally determined parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 4096$, $h_1 = 6%$, $h_2 = 14%$, $h_c = 10%$</td>
<td>$k_r = 9$, $\beta = 1.44$, $\tau = 5$, $\tau_{\text{majority}} = 3$</td>
</tr>
</tbody>
</table>

In summary, for LAWN the set up consists of recognizing token lengths $n = 4096$ with the cut-off thresholds $h_1 = 6\%$ and $h_2 = 14\%$. By doing the pre-deployment tuning, we arrive at the desired value of cut-off $k=9$ that works well for this setting. We fine-tuned the system parameters further to reduce the number of false positives. Table 5 summarizes the outcome of the experiments. Once $k_r$ and $\beta$ are determined, the deployment of LAWN is straightforward. Note that once set, these parameters don’t change in the real deployment unless there is a change in the input specifications.

4.5 Evaluation of LAWN

In this section we assess LAWN on other metrics, viz., the additional time it incurs and also the estimated power gain. The evaluation of the tuning methodology establishes confidence that LAWN is quite secure to be used in practice. In this section we support the usability of LAWN by measuring its overheads and advantages.

4.5.1 Time Overhead of LAWN

LAWN trades computation for communication. Although the benefit related to network activities is obvious, we need to validate that the extra computation does not add significant time overhead in a realistic setting. Thus we ported the prototype implementation of LAWN over an iPAQ 3870, (Intel StrongARM 100 processor, 200 MHz, with an ORiNOCO PC wireless card on IEEE 802.11b, familiar Linux distribution). A desktop PC (Intel x86, 660 MHz, 512 MB RAM, Redhat Linux 7.1) serves as the back-end verifier.

Authentication strings are sent every 10 seconds. We examine two scenarios. Naively transmitting the full tokens, and sending a small sketch using LAWN. Figure 32(a) shows the time taken by each process as a function of increasing size of the token. Compared to the naive method, LAWN does take additional time. However, for a string of 16 KB, we notice that the absolute time taken by LAWN is only 300 ms, as opposed to 190 ms taken.
by the naive method. It is not a significant overhead even considering a fresh authentication every 10 seconds.

Next we tease out the time spent by LAWN into computation and communication. Figure 32(b) shows how this split is distributed across increasing token size. Although the computation time is increasing, the time spent on the network link remains almost constant. The reason is that the we have chosen a fixed length of a sketch (30 bytes) that is sufficient for LAWN to distinguish strings of all the string lengths considered in this experiment (this length should be a little more than the expected cut-off k, to separate good and bad strings properly). The communication time is significantly lower than the communication time taken by the naive method (from Figure 32(a)). For example, on a 16 KB string, while the naive methods spends around 175 ms on the network link (no computing is involved), LAWN spends only around 50 ms. Thus we clearly see that network activity for LAWN is substantially lower. Moreover, the time spent in networking is not directly proportional to the number of bytes transferred; in reality, although the time spent by LAWN in networking is almost one fourth of the naive method, the number of bytes transferred is far less and thus the saving in power is expected to be even higher.

4.5.2 Power Consumption Analysis

In this section we provide an analysis of the amount of power savings that can be achieved by deploying the proposed scheme. In general, such estimation can be quite complicated, the consumption of power being a function of many factors such as processor architecture (including the CPU and memory subsystems), the network link that involves analog components and also software factors such as the operating system, exact implementation of systems libraries and so on. Instead of stressing on fine-grain precision, we evaluate our figure of merit in a rather coarse-grain way. The main power-drawers involved in this scenario are the processor, memory, and the network link. We compare two alternative methods; sending the whole data block naively over the wireless, and sending the hamming-distance sensitive hash-vector instead of the data block. Since, some computation is involved in the second case, we have to estimate the power consumed by the processor. The processing
power consists of two parts: the energy consumed at the CPU by the instructions and the energy spent due to the cache-miss and memory accesses. Since the additional memory requirement of our scheme (over the baseline case) is very small, essentially just the size of the hash vector that would easily fit into the local cache of any modern processor, we can assume that the additional power consumption related to memory/cache can be kept out of the consideration and that we can solely focus on instruction level power analysis[87]. According to this scheme, energy consumed by each instruction is measured in isolation and then the total power is evaluated from the profile of the machine code. The consumption naturally varies from one instruction to another, however, one can consider an average case or worst case picture.

Table 6: Parameters relevant for power-analysis

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>original size of the key</td>
</tr>
<tr>
<td>$e_{\text{proc}}$</td>
<td>average energy for executing one instruction</td>
</tr>
<tr>
<td>$e_{\text{net}}$</td>
<td>average energy to transmit one bit</td>
</tr>
<tr>
<td>$N_i$</td>
<td>number of instructions needed to compute one hash vector</td>
</tr>
<tr>
<td>$b$</td>
<td>bits needed to represent one digest</td>
</tr>
<tr>
<td>$t$</td>
<td>number of random samples required for one sketch</td>
</tr>
<tr>
<td>$v = t.b$</td>
<td>size of one hash vector</td>
</tr>
<tr>
<td>$\tau$</td>
<td>number of repeated trials</td>
</tr>
<tr>
<td>$I = \tau.N_i$</td>
<td>total number of instructions</td>
</tr>
</tbody>
</table>

Table 6 describes the parameters of interest for computing the power consumption. The total power consumption for the first case (sending the whole key) is given by $E_1 = n.e_{\text{net}}$. In the second case, there are two factors, processing and transmission, and the total energy is given by $E_2 = \tau.v.e_{\text{net}} + I.e_{\text{proc}}$. Hence, the ratio of energy consumption in two cases is given by

$$\frac{E_2}{E_1} = \frac{v\tau}{n} + \left(\frac{I}{n}\right)\left(\frac{e_{\text{proc}}}{e_{\text{net}}}\right)$$

Notice that $v = \mathcal{O}(\log n)$, and the fraction $v.\tau/n$ decreases sharply as the size $n$ of

---

2Although there is some minimal processing required, such as byte copying, we assume the power is negligible. If anything such assumption favors underestimation of the savings that we predict.
the data block increases. For example, for a data size of 4Kb, hash bucket size \( b = 4 \) bits, \( \tau = 5 \) repeated trials, and a hash vector length \( t = 20 \), the length of the transmitted hamming sketch is \( v.\tau = t.b.\tau = 400 \). Pottie and Kaiser [72] estimate that transmitting one bit wirelessly over a distance of about hundred meters is equivalent to 3000 instructions on a standard processor, which means \( e_{\text{proc}}/e_{\text{net}} \approx 1/3000 \). Although this rough estimate is slightly outdated, the scenario has changed only in favor of low power processing; while long range transmission is limited by the physical factors beyond control, modern processors are becoming increasingly power efficient. At any rate, assuming \( e_{\text{proc}}/e_{\text{net}} = 1/3000 \), and noticing that it takes about 30000 instructions\(^3\) to compute five sketches, we get

\[
\frac{E_2}{E_1} = 0.1 + \left( \frac{30000}{4000} \right) \left( \frac{1}{3000} \right) = 0.10025
\]

As \( n \) grows, the factor \( E_2/E_1 \) keeps constantly decreasing. As we have already seen in section 4.4.1, the total bandwidth and power savings actually depend on the false negatives; every time a legitimate string is rejected as false, the whole key is communicated in order to convince the verifier at the remote end. The fraction of such false rejections, as depicted by Figure 30(b) is less than 20%. Hence, amortized over many trials, the total energy ratio is given by

\[
\frac{0.8E_2 + 0.2E_1}{E_1} = 0.2 + (0.8)(0.10025) \approx 0.28
\]

Therefore, amortized over many transmissions, such a scheme results in bandwidth and power savings up to 72%.

As we mentioned before, with increasing size of the key, we can see more savings in power. Consider a key size of 8Kb. Since the size of the sketch need not change (only \( \beta \) needs to be increased), plugging in appropriate numbers into the previous formula for energy-ratio we observe that \( \frac{E_2}{E_1} \approx 0.005 \) and this will lead to an energy ratio \( \frac{0.8E_2 + 0.2E_1}{E_1} = 0.2 + (0.8)(0.05) \approx 0.20 \) The above estimate tells us that for a string of 8Kb, the savings can be almost up to 80%. However, with a false negative rate of 20%, this is the maximum

\(^3\)by examining the machine code of LAWN’s sketch-computing algorithm on an x86 processor
savings possible. Further enlargement in the key size will bring the savings asymptotically close to 80%.

**Power consumption on 802.11b platform**

We now present the power estimates on the short-range wireless transmission. Although the benefits are not expected to be as high as it would be in a very long-range scenario, the evidence is clear that LAWN would still bring substantial gains in power consumption.

We took a power profile of the LAWN implementation on StrongArm Linux (presented in 4.5.1) with the help of the JouleTrack [81] tool. JouleTrack is a web based energy profiling tool for StrongARM SA-1100 processor. This tool gives us an estimate of the power consumed by the computing cycles. For communication, we used numbers corresponding to OriNoCo network card [2] operating at 2Mbps. LAWN is compared with naive authentication (sending the whole key) over different key sizes.

Figure 33 compares the total power consumed by the LAWN and naive authentication. The detailed breakdown of these numbers (not included herein to keep the presentation simple) shows that for LAWN, the communication component of the energy is less than 1mJ and remains the same with increase in token size. However, the computation power consumption increases linearly as token size increases, but since communication is more energy consuming than computation, we observe savings in total power consumption. At 206MHz operating frequency, LAWN consumes 3 times less power compared to the brute force authentication. Bringing down the operating frequency, though increases the computation time, saves more computation power. For example, at 59MHz operating frequency, LAWN takes 80mSec more time to compute the sketches than at 206MHz, but it consumes about 10 times less power than the brute force authentication.

The above results are for one instance of authentication. However, amortized over a large number of authentications in presence of false negatives, the power gain is slightly reduced as already explained. For example, for 4KB token size, the energy ratio $E_2/E_1$ is approximately 0.1 at 59 MHz operating frequency and 0.35 at 206 MHz. Thus the amortized energy ratio with 20% false negatives ($\frac{0.8E_2 + 0.2E_1}{E_1}$) becomes 0.28 and 0.48 respectively, which
means savings of 72% and 52% of power over naive authentication.

4.6 Related Work

A near future reality of having biometric authentication in bandwidth and power-limited mobile devices serves as the motivation for the work presented in this paper. Although we do not deal with any explicit biometric algorithm, we believe the simple technique can be extended to such cases. Moreover, computing Hamming distance is itself an important component in many pattern recognition algorithms. For example, the Daugman system [27], a commercially used retinal scan technology, uses Hamming distance in the following way. Once a retinal image is taken by the scanner, a series of subsequent transformations finally produces an Iris Code which is essentially a flat bit string. Two retinal images can be compared by checking the Hamming distance between their iris codes. Typically iris codes from two different bit strings differ in more than 50% bits, while the codes produced from the same iris image (taken on different occasion) are seen not to vary on more than 10% of the bits [16]. Similarly, some face recognition algorithms convert a face image into a feature vector so that any subsequent comparison is reduced to Hamming distance computation between such strings [45]. Fingerprint verification algorithms follow similar route.

With the observation that remote authentication is nothing but a distance computation, our concern of minimal network usage reduces to the communication complexity of the distance function. Communication complexity [51] is a field in theoretical computer science that deals with the following question. Suppose Alice and Bob are two parties trying to compute a function \( f(x, y) \). Alice has input \( x \) but not \( y \), and similarly Bob has \( y \) but not \( x \). What is the minimum number of bits necessary to be communicated between them so that they both can compute \( f \).

Our problem, in an abstract sense therefore resemble that of the probabilistic communication complexity of Hamming Distance computation. One restriction however, is that it should be secure in at least one way, i.e., the end-user (who can be potentially an adversary) should gather in the process no knowledge of the template stored in the remote data-base.
Secure protocols for two way estimation of Hamming distance is discussed in [30] which also provides a simpler protocol for estimating Hamming distance. However, the bounds are not very sharp. The scheme used for LAWN has its roots in Hamming distance based clustering proposed by Andersson et al [8]. The sampling technique was borrowed from the works of Cormode et al. [22]. Our contribution lies partly in identifying the relevant theoretical insights and then adapting them into a protocol and proving its high-probability guarantees, and partly in designing the experimental methodology to bridge the gap between theory and practice.

A crucial ingredient of our paper is the use of Error-Correction Code (ECC). In addition to almost ubiquitous use of ECC in digital communication and storage, it has found many fundamental applications in techniques and results of Computational Complexity Theory [56]. In particular, ECC has been used in Probabilistic Checking of Proofs [9], Holographic Proofs [83, 71], and Communication Complexity Theory. Error correcting codes have significant implication in security protocols; one very relevant application that is especially related to biometric authentication can be found in [43].

Adaptation related projects [31, 28] have shown how to change application behavior to save power in mobile scenario. LBFS [69] shows techniques to avoid transmitting duplicate data to save communication overhead. These research findings have motivated us to look into the estimation of how much power can be saved by adopting a randomized scheme. It is however fairly difficult to quantify the exact power consumption, as it typically requires hardware instrumentation and kernel level support [32]. For some recent results on low power communication the reader is referred to a dissertation by Min [66]. Data on energy dissipation in some of the commercial processors and network links are reported by Fryman et al. [33] and similar information but at the level of instruction execution can be found in some of the documents related to Instruction Level Power Analysis [87]. Although our intuition is based on the conventional wisdom “compute more and communicate less”, very recently this wisdom has been challenged by some researchers, mostly arguing in the line that architectural features in modern processors hinder the power savings expected from the reduction in communication [33, 10]. For example, Barr and Asanovic [10] show that
compressing files before releasing it onto the wireless, may end up increasing the energy
needs, due to various factors such as the DRAM to cache ratio, cache miss rates and so on.
However, the size of caches in modern processors are big enough to hold our authentication
tokens and we thus we feel safe to assume that there will not be too many cache misses to
add on to LAWN's power consumption.

4.7 Conclusions

Remote Authentication over wireless is one of the key enablers for many current and future
applications. We have presented LAWN, a light-weight power-aware protocol that is based
on the conventional wisdom compute more, communicate less. We provide the design and
the analysis of the theoretical guarantees for this system. We have also built a prototype
system and provide its performance evaluation over iPAQ platforms. A crucial step in
getting such a system work is to carry out a series of preprocessing before its deployment.

We describe the experimental methodology to do so and evaluate the methodology as well.
So far, our results are limited to approximating Hamming distance. However, even this
limited result is important for two reasons. First, Hamming distance computing is at the
core of many pattern matching algorithms that are used in authentication schemes. And
second, a generalization for any kind of metric is likely to exist.
Figure 29: Variation of expectation and the variance of observed value of $k$ with the Hamming distance. For these experiments the string length was chosen as $n = 4096$ and $\beta = 1.3$
(a) False positives & negatives as a function of cut-off $K (k_c)$

(b) Accepts and Rejects after five repeated trials. Y-axis measures the probability (normalized from the experimental numbers) of accept or reject

Figure 30: Determining $k_c$ and results after repeated trials, $n, \beta$ same as before
Figure 31: Variation of $h_1$ and $h_2$ as a function of $\beta$ keeping other parameters constant, $n = 8192$
(a) Time overhead of LAWN

(b) Computation-communication split of time in LAWN

Figure 32: Results on an iPAQ
Figure 33: Power consumption profiling for LAWN and brute force authentication (sending complete token).
CHAPTER 5

CONCLUSIONS

Error-correcting codes can play multiple roles in designing secure and robust distributed and pervasive systems. This thesis bears testimony to this fact by presenting some results in the intersection of Security and Fault Tolerance for these environments.

First, We discussed a new design principle, i.e, that of using error correcting code based approach for building secure, fault tolerant storage. The design of SAFE exploits this principle. We showed that a secure distributed storage can be implemented without significant performance cost compared to other alternatives.

Next, We discussed the design of e-SAFE, a distributed storage service targeted for very large scale decentralized storage. In the design of e-SAFE we made a quantitative observation, that of equating high load with low availability. At the heart of e-SAFE is a special class of codes, called the Fountain codes, that makes e-SAFE seamlessly adaptable to unlimited stretching and thus to hardware extensions of any degree.

Remote Authentication over wireless is one of the key enablers for many current and future applications. LAWN is a light-weight power-aware protocol that is based on the conventional wisdom compute more, communicate less. We presented the design and the analysis of the theoretical guarantees for this system. We have also built a prototype system and conducted performance evaluation over handheld platforms,
CHAPTER 6

FUTURE WORK

The design of SAFE serves as a launching pad for the development of an end to end file system, which is currently being investigated. From a cryptography point of view, one needs to analyze in detail the practical security of such systems by assuring that there are no weaker keys (or if any, how to set the system’s parameters so as to avoid them) and possibly doing adequate cryptanalysis. Thus by identifying a fresh body of application, we hope to stimulate further research in the linear-code based crypto-systems.

SAFE has a novel design in using modern codes (the fountain codes) and verification techniques. It is targeted to cater to very large and complex storage servers. As part of the ongoing and future work, we are investigating the dynamics of workloads more closely and the resulting performance of SAFE. Diverse loads and complicated asynchronous behaviors of various components leave open a plethora of questions that can only be answered by combining more analytical studies such as random processes and queueing theory techniques with our experimental methods. Such a study is part of our future research.

In the study of LAWN, we have discussed the problem of remote authentication over long range wireless networks. So far, our results are limited to approximating Hamming distance. However, a generalization for any kind of metric is likely to exist; one can hash patterns into spheres of exponentially decreasing size (the sphere being determined by the specific distance in question) and thus prepare a sketch for the pattern in question. However, this requires further investigation, both theoretical and experimental.

In this work we mostly analyzed the benefits that LAWN can bring to a resource-strapped client. A further implication can be on the load handling aspect of the remote server (verifier). While using LAWN, the server’s computation is somewhat reduced as well, since LAWN requires only approximating the Hamming distance instead of computing the same. Plus, network I/O is somewhat reduced for the same reason. Thus, given a single
server that handles a set of authentication requests at a high intensity, the performance-
impact of LAWN on the server may be beneficial to a non-trivial extent. We intend to carry out that study as part of our future work.
Appendix

Proof of Theorem 1

Table 7: Variables used in the analysis of Theorem 1

<table>
<thead>
<tr>
<th>$h$</th>
<th>hamming distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>hash collision probability</td>
</tr>
<tr>
<td>$X$</td>
<td>the proof sting</td>
</tr>
<tr>
<td>$Y$</td>
<td>the template string</td>
</tr>
<tr>
<td>$X_l$</td>
<td>$l$-th sample chosen from the proof string</td>
</tr>
<tr>
<td>$Y_l$</td>
<td>$l$-th sample chosen from the template string</td>
</tr>
</tbody>
</table>

We now prove Theorem 1. Table 7 lists the variables necessary for this analysis. We estimate the bounds on the estimation of $h$. Note that, given two bits from the same position of the two strings, the probability that they are different is $h/n$. Hence, for any independent sample $l$, (of length $\beta_l$), the probability that the $b$-th bits of the template-sample and proof-sample differ, is $h/n$. This is simply because the two strings differ in $h/n$ fraction of bits. Hence we have the following set of equations:

$$Pr[(X_l)_b \neq (Y_l)_b] = h/n \quad \forall 1 \leq b \leq \beta_l$$

$$Pr[(X_l)_b = (Y_l)_b] = 1 - h/n$$

Setting $\delta = (1 - h/n)$, we get $Pr[X_l \neq Y_l] = 1 - (1 - h/n)^{\beta_l} = 1 - \delta^{(\beta_l)}$. Thus $Pr[X_l = Y_l] = \delta^{(\beta_l)}$. This is an approximation. Alternatively, this probability can be obtained by noticing that $Pr[X_l = Y_l] = \binom{n-h}{\beta_l}/\binom{n}{\beta_l}$ For large $n$, the approximation of this hypergeometric distribution becomes $(1 - h/n)^{\beta_l}$.

Now we estimate the error probability. $\mu$ is the probability of collision in the case of hashing $H(x)$. We first estimate the probability of having a collision in hashing for two distinct samples of length $\beta_l$. This is simply given by the conditional probability $Pr[H(X_l) = H(Y_l)|X_l \neq Y_l]$. Now, observe from Bayes’ Theorem :

$$Pr[H(X_l) = H(Y_l)|X_l \neq Y_l] = \frac{Pr[H(X_l) = H(Y_l) \cap X_l \neq Y_l]}{Pr[X_l \neq Y_l]}$$

Let’s assume that the first difference in the two hash-vectors is observed after comparing $k$ positions. Hence, we estimate the probability that first $k$ positions produce no difference. Note that
the numerator on the right hand side indicates the probability of not catching any difference in the
l-th sample. The left hand side is nothing but the collision probability \( \mu \) and the denominator is
\( 1 - \delta^{(\beta^t)} \). Plugging in these values, the probability of not catching any difference in the
l-th sample is given by \( \mu(1 - \delta^{(\beta^t)}) \). Now observe that the first \( k \) locations of the sketch vectors could match in two
ways. Either \( X_i = Y_i \) and thus they produce the same Hash value, or, \( H(X_i) = H(Y_i) \cap X_i \neq Y_i \).
Since these two are mutually exclusive events the probability that all the first \( k \) positions of the
feature vectors match is given by \( \prod_i \delta^{\beta^t} + \mu(1 - \delta^{\beta^t}) \). We set this probability more than \( 1 - \epsilon \) which yields
\[
\prod_i \delta^{\beta^t} + \mu(1 - \delta^{\beta^t}) \geq 1 - \epsilon
\]

In \( \mu \) is reasonably small, one can approximate \( 1 - \mu \) with 1, setting \( \lambda = h/n \), using a first order
Taylor series approximation and further simplification the inequality becomes:
\[
\prod_i [1 + \mu - \lambda \beta^t] \geq 1 - \epsilon
\]

Simplifying the above and putting \( \phi = \frac{\lambda}{1+\mu} \) We get
\[
(1 + \mu) \prod_i (1 - \phi \beta^t) \geq 1 - \epsilon
\]

Further approximating the product terms, we get,
\[
(1 + \mu)(1 - \phi \beta)^k \geq 1 - \epsilon
\]

The rest can be simplified from the above relationship.
REFERENCES


98


99