AN ARCHITECTURE FOR SECURE SEARCHABLE CLOUD STORAGE

a dissertation
submitted to the Department of Computer Science,
Faculty of Science
at the University of Cape Town
in fulfillment of the requirements
for the degree of
Master of Science

By ROBERT KOLETKA
FEBRUARY 2012

Supervised by
Andrew Hutchison
and
Co-Supervised
Sonia Berman

Acknowledgements

First and foremost I would like to thank my Masters supervisor Dr. Andrew Hutchison for his guidance, insight, knowledge and help in completing this project. I would also like to thank Dr. Sonia Berman for providing co-supervision assistance with the project. Thank you to my fellow students in providing any support and discussion with regards to this project. I would also like to thank Benjamin Tobler for helping me with my initial project idea without breaching any Non-Disclosure agreements he may have had with Amazon.com. Lastly I would like to thank my family for their continued support and enthusiasm for the duration of this dissertation.

Abstract

Cloud Computing is a relatively new and appealing concept; however, users may not fully trust Cloud Providers with their data and can be reluctant to store their files on Cloud Storage Services. The problem is that Cloud Providers allow users to store their information on the provider's infrastructure with compliance to their terms and conditions, however all security is handled by the provider and generally the details of how this is done are not disclosed.

This thesis describes a solution that allows users to securely store data on a public cloud, while also providing a mechanism to allow for searchability through their encrypted data. Users are able to submit encrypted keyword queries and, through a *symmetric searchable encryption scheme*, the system retrieves a list of files with such keywords contained within the cloud storage medium.

Securing distributed storage typically falls into two major categories one being where the client trusts the storage medium; and the other where all trust is removed and the storage medium is assumed to be unsecure. The secure cloud storage system described is designed in such a manner that trust from a public cloud provider is not required. This is achieved by adapting techniques used in securing distributed storage where the storage medium is not trusted. The solution satisfies *confidentiality* of data from the cloud provider or any other third party; data *integrity* can be checked; file sharing amongst users is catered for and a user key management and *key-revocation scheme* is in place, together with the ability to search for relevant files. These requirements are introduced and developed in this research. A further advantage of the proposed approach is that if there is a security breach at the cloud provider, the user's data will continue to be secure since all data is encrypted. Users need not worry about cloud providers gaining access to their data illegally

nor whether their data may be stored in 'foreign' jurisdictions, where the provider could be forced to reveal data by a court order.

The architecture of the system presented consists of two components, a *Client side application* and a *Server application* running on the compute cloud. The client side application performs all the cryptographic operations on the data. Along with saving and retrieving data from the Cloud Storage Service, the server application performs the processing involved in handling the encrypted queries by running on a virtual machine instance within the compute cloud. It performs these operations by using the search algorithm that is adapted from the symmetric searchable encryption scheme. The solution adds overheads in terms of additional processing time and the size of the additional meta-data needed, but this is considered tolerable considering the security functionality added.

The design of the solution is described in detail in the document, discussing the data structures needed to keep data secure within a cloud storage provider, as well as the data structures and algorithms needed to authenticate with the server. We discuss the adaptation of the symmetric searchable encryption scheme, documenting the data structures that are needed for this functionality as well as the algorithms that will be necessary to perform the searching as well as the secure keyword generation.

Functional testing was performed to ensure that the design of the system could be correctly implemented and satisfies the design requirements. After testing the prototype at a functional level, it was important to measure the impact on performance and storage overheads of the design. This was achieved by testing the time the prototype took to secure files of varying sizes, examine the storage overheads in securing those files, the difference in uploading files securely and unsecurely and the time taken to execute search query at both a client level and a server level.

In examining the results that we have gathered in our testing, we have seen that our prototype imposes a minimal overhead. Our testing has shown that the maximum meta-data overhead for our test cases was 910 Bytes. Since our overheads are insignificant, the impact on uploading unsecured data versus secured data is also minimal. When testing the searching functionality of our prototype we considered the response time at the client, as well the the performance of our search algorithm at the server where we have eliminated all network latency.

It was equally important to show that this solution could be applied in a real setting. To demonstrate this we modified Alpine, a Linux emailing application, to interface with our prototype and upload emails with keywords to the cloud storage service. We demonstrated this by creating a simple Python script that sends commands via a socket to the client. The client then executes the commands and send the Secure File Object to the server. This was all achieved with minimal effort and shows that this solution has real world applications.

This research shows that it is possible to securely store confidential user data on a Public Cloud such as Amazon S3 or Windows Azure Storage without the need to trust the Cloud Provider. It is also possible to upload secured data while giving users the ability to search and retrieve only the encrypted files that they need. All this can be achieved with a minimal overhead in both processing time as well as storage.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Storing information securely implies that the information is *confidential* and is only accessed by legitimate users, that there are mechanisms in place to ensure data *integrity* and that consideration is given to *authentication* and *access control*. There have been concerns about the security and trust with regards to Public Cloud Providers such as Amazon's Web Services[1, 17, 19, 10].

1.1 Motivation

There exists a vast amount of infrastructure that is made available by Cloud Providers, that can be used to store information and perform processing, however research by Armbrust et al.[1] show that users are very hesitant when it comes to storing private information on a service such as Amazon's Simple Storage Service.

There has however been substantial research into securing distributed systems which is discussed further in Chapter 2. There is also a usability problem in retrieving only the relevant encrypted information for processing, especially when there are network speed issues.

There was a study performed by Hacigümüş, H et al. [13] to develop a means to query a relational database. The aim of this study, as discussed in Section 1.2, will be to apply similar principles in a non-relational setting.

1.2 Objective

The aim of this research is to design a solution that provides users with a means of securing their data without neccessarily trusting a cloud storage service, and analysing the performance implications.

In doing this, user data will not be compromised should the storage provider be compromised in any way (either internally or externally) . The system will need to satisfy a set of requirements as specified in Chapter 3 in order for it to be deemed secure.

The dissertation also examines techniques that can be used in providing a means to search through encrypted user data, in such a manner that the cloud provider is unable to gather any information from the queries and the results but the user has a selective retrieval mechanism. By doing this users are able to retrieve only information that is relevant to their search.

1.3 Evaluation

The hypothesis of the study is that *secure searchable storage can be securely incorporated into in a cloud storage service.*
In order to evaluate the hypothesis we will need to evaluate if the approach satisfies all the requirements of a secure storage system as specified in Chapter 3. We will also need to examine the performance overheads of this approach and compare them with that of similar systems to decide whether the overheads of our approach are satisfactory. More specifically we will examine:

1. The overhead of securing data and storing it on a cloud storage service.

2. The overhead of adding search functionality for a data stored securely on a cloud

storage service.

1.4 Assumptions

The project assumes that the cloud provider is not trusted and therefore all cryptographic operations are done on the client. The client computer is also assumed to be secure. The file owner gives access to users that can be trusted however the implications of malicious users will be discussed. The system only deals with a *small* number of files, in the order of hundreds, since the purpose of this study is to examine whether secure cloud storage can be achieved and not neccessarily how scalable it is, although this is also considered to be favourable and will be re-visisted in Chapter 8 on future work. There are various techniques that can also be explored to optimize the search process, however these are out of the scope of this project. We also assume that only the file owner is allowed to generate search capabilities for users who wish to search through the encrypted data.

1.5 Approach

To achieve the objective and meet the requirements we take the following approach.

- Analyze the requirements of a secure storage system. We need to understand what needs to be fulfilled in order for a system to be deemed secure.
- We will to analyze techniques used to for providing searchable encryption and decide which one is most relevant to our setting.
- We will then combine and adapt the techniques used in securing storage systems, and incorporating searchable encryption into a systems design that satisfies the requirements and objectives.
- Building of a prototype and testing will be necessary, to evaluate the success of the project.

- Deployment of the secure storage system in a widely used appliaction. For this purpose cloud deployment of the folders of an email system will be tested.

In following this approach it is our goal to realise a system that can securely authenticate with a system running with the cloud that accepts requests and queries. Our system is to secure all data before it is dispatched to the cloud. The system running within the clouds sole purpose will be to peform the search operations and no other functionality since the assumption of this study is that cloud providers are untrustworthy thus all security operations are performed by the client.

1.6 Dissertation Outline

The structure of the dissertation is as follows:

- In Chapter 2 we provide background information into cloud computing and the techniques behind scalable storage as is used by cloud providers. We then discuss certain methods that are used to secure distributed storage system and lastly we discuss the idea of searching through encrypted data.

- Chapter 3 discusses the requirements that need to be satisfied in order to secure a storage system.

- The design of the approach, to satisfy the requirements, is described in Chapter 4. This chapter discusses the data structures and algorithms needed by our approach as well as the design of the architecure of the system.

- We then document how the prototype was implemented in Chapter 5.

- Chapter 6 describes the testing that was performed on our prototype and documents all the information that was gathered. We test our prototype at both a functional and performance level to help us evaluate the success of this approach.

- After testing our prototype we analyze the results and findings in Chapter 7.
- We conclude our research in Chapter 8 by discussing the approach and discussing any future work that can come from this research.

Chapter 2

Background

This chapter introduces the ideas behind *Cloud Computing* and how it can be used. We will discuss the three main levels of abstraction present in the cloud computing model as well as introduce the idea of *Cloud Storage-as-a-service* and describe two prominent cloud storage services. It is also important to look into how one communicates with such a cloud storage service, namely via *REST* and *SOAP*; as well as to understand how general *Distributed Storage Systems* operate with respect to Cloud Storage. We will then review a few *Secure Distributed Systems* as examples of techniques that are relevant in designing secure cloud storage systems. Finally, we discuss recent work on *Encryption with Keyword Search.*

2.1 Cloud Computing

Cloud Computing is the emerging model where functionality is exposed as a service over a network, this refers to both Applications and Hardware. It has been generally accepted that there are three levels of abstraction within this paradigm namely; *Software-as-a-Service (SaaS) , Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service(IaaS).*[22] SaaS means that Software is being exposed as a service over a network such as Google Doc's. PaaS allows for a software platform to be hosted over the network such as Microsoft's Azure or Google App Engine. IaaS is the lowest level of abstraction and at this level virtual machines (VMs) are exposed as a service over the network, giving users the most flexibility as in the case of Amazons EC2[1][22].

[1]Elastic Compute Cloud

The advantage of using a cloud provider is that the users need not worry about the complexities of managing a data-center. For example, when using Amazon's EC2 service, the user is simply given an EC2 VM instance and is not concerned about where the instance is or whether is will still be available during network partitions within Amazon's data-centers. Additionally using a Cloud Provider, a user only pays for what they use. This is an advantage since users do not need to invest large amounts of money into a data center that will only be fully utilized during usage spikes, which may occur only a few times within a year, whilst being underutilized the rest of the time.

Storage as a service is a way of storing data in a cloud in a manner that the data storage and replication is transparent to the user. Details of Cloud storage are discussed in Section 2.2.

2.1.1 Cloud Architecture

Lenk et al. [22] describe an architectural landscape where there are different levels of abstraction nested in one another. For example SaaS sits on top of PaaS which in turn sits on top of IaaS. One could think of Google Docs running on top of some platform service which in turn runs on top of a virtual infrastructure service.
Cloud Computing Systems such as Open Stack[2], Nimbula[3] and Zimory[4] all follow a similar architecture within their virtual data center environments. The key focus of Cloud Computing is abstraction, this allows the users to use the services without focusing on the complexities of the underlying infrastructure.

[2] http://www.openstack.org/
[3] http://nimbula.com/
[4] http://www.zimory.com/

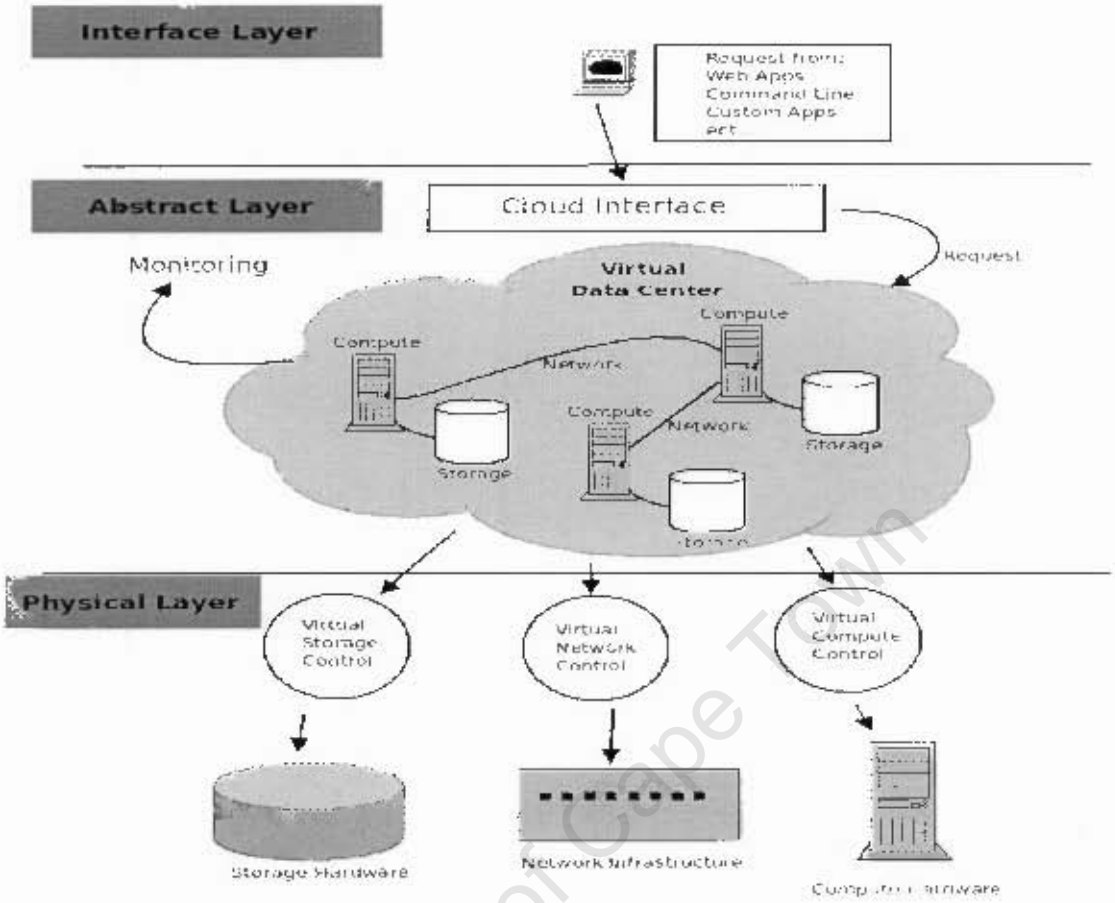


Figure 2.1: Architecture a Cloud Computing Environment

As shown in Figure 2.1, a Cloud Computing Environment can be seen to consist of an Interface layer, an Abstract layer, and a Physical layer. Users of the cloud interact via the Interface layer. The interface then communicates with the virtual data center. The virtual data center is an abstraction of the physical hardware, exposing a generic compute instance and a generic storage API. This can be compared to Amazon's Elastic Compute Cloud and Amazon's Simple Storage Service. This diagram is a general representation of the abstractions within a cloud architecture such as Amazon.com, Nimbula and Open Stack.

2.2 Cloud Storage

A cloud storage service is a way of storing data within a cloud environment. This means that a user submits one copy to the storage service via an interface as described in Section 2.5.1, and all the replication and physical storage is handled by the storage service provider. The drawback with such a storage service is that all security is handled by the storage provider. Services such as Amazon S3[5] and Microsoft's Azure Storage provide the cloud users with authentication keys that allow the users to access an object given a key. Offloading security to the storage provider has both advantages and disadvantages. An obvious advantage is that the user doesn't need to worry about the complexities of securing the data. However a disadvantage is that if the security of a storage service is compromised, then some user data may compromised which can have devastating effects if user data is confidential as was the case with LinkUp(MediaMax), where the company went out of business after losing 45% client data[6].

We will now discuss Amazon Dynamo which is a key-value storage system used by Amazons internal systems as well as the Google File System .

2.2.1 Amazon Dynamo

Amazon's Dynamo is a highly available key-value store that is used by Amazon's platform. Dynamo provides a Primary-Key only interface which allows the user to submit a key and return a value. This is the type of service that is of relevance for our Secure Storage System, such as envisaged in this thesis for addressing the security concerns when storing data within a public key-value store. Dynamo is an internal Amazon service, however Amazon's S3 is also a key-value store, this will be used when testing our Secure Storage System.
Data within Dynamo is partitioned and replicated by using consistent hashing[18]. Dy-

[5]Simple Storage Service

[6]`http://blogs.zdnet.com/projectfailures/?p=999`

namo replicates data on multiple servers. Since dynamo uses an asynchronous replication and update scheme, the replicated data will be eventually consistent, implying that inconsistencies can occur and it is up to the application to resolve them. Consistent Hashing, partitioning and asynchronous updates allow for Dynamo to be completely decentralized, highly available and scalable.

2.2.2 Google File System

The Google file system(GFS) was designed and implemented to be a scalable distributed file system. The system was built to function on commodity hardware, thus it was designed to be fault tolerant due to the assumption that commodity hardware tends be faulty.

A GFS cluster consists of a single *master* and multiple *chunk-servers*. A file is broken up into a number of chunks that are stored and replicated across a number of chunk-servers. The master maintains all the system meta-data such as name-space, access control, mapping of files to chunks and the physical location of the chunks. The master is simply there so as to maintain system state and route client requests to the correct chunk-server so as to not cause a bottleneck.

As shown in Figure 2.2, a typical interaction would be that the client connects to the master. The master then looks up the physical location of the chunk and then returns this address to the client. The client then connects to the correct chunk-server and requests the data. By doing this, the load is taken off the master.[11]

2.3 Distributed Storage

A distributed storage system allows client computers to access files over a network. This can be used to achieve higher reliability, by storing multiple copies of the data. If there are network outages in certain parts of the storage network then data can still be retrieved from other parts of the network. Examining distributed storage system is relevant since

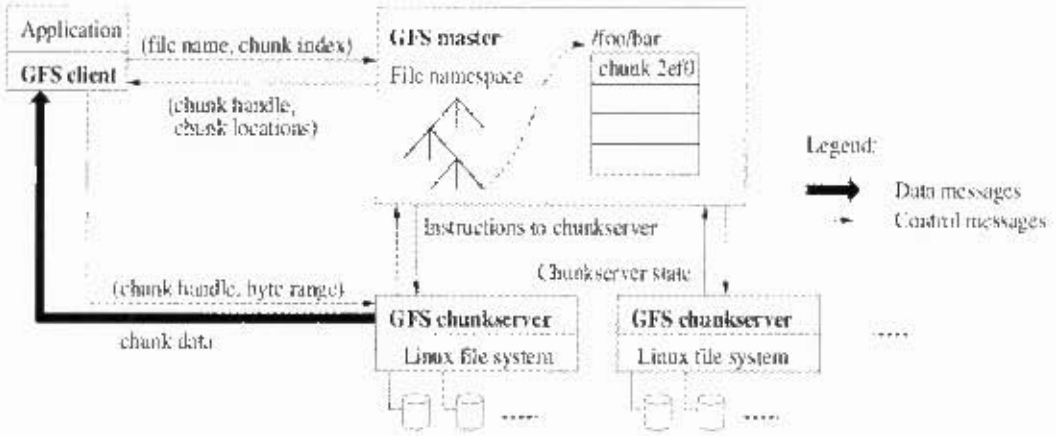


Figure 2.2: Architecture for the Google File System

such techniques are needed by the cloud providers at the physical layer in Figure 2.1.

Traditionally remote storage was achieved by using systems such as NFS[26] and AFS[14]. The problem with such systems is that they are centralized, causing a bottleneck and a single point of failure.

For demanding systems in the Cloud Domain, a storage system must be available all the time[7]. Such demanding requirements have been solved for the most part by using Peer-to-Peer(P2P) storage systems. Such systems are decentralized and thus there is no single point of failure or any bottlenecks. A centralized P2P network uses an indexing server which could cause a bottleneck or system outage should it fail, for these reasons decentralized P2P networks are favored.

A P2P storage system is made up of a number of storage nodes. These storage nodes

[7]As in the Amazon's case, this implies at least 99.99% up-time during any monthly billing cycle as stated in their S3 Service Level Agreement

are used to store the data as well as route requests and handle data replication. One of the challenges of using P2P is to provide efficient algorithms and techniques for object location and routing[25].

The simplest but most inefficient method of object location would be to query every node in the network for a given object. A common technique used by Chord[29] and Pastry[25] is to use a Distributed Hash Table[8]. A DHT uses an abstract key-space derived from the Key-ID of a storage node of object. This abstract key space is then partitioned in various ways so as to split ownership of this key space among the nodes. This DHT is then used to route the data towards the node that is in control of the sub key-space where a query falls in. Such techniques are used by Amazon's Dynamo[9] and Cassandra[21][9]. This is far more efficient than the brute force method of querying each node, as it does not flood the network with queries.

2.3.1 Consistent Hashing Example

As discussed distributed storage systems need techniques for storing data efficiently and route requests to the correct storage nodes correctly. Consistent Hashing is a techniques that is used by the Amazon Dynamo system to ensure that data is available and replicated and that requests get routed to the correct storage nodes with the least amount of overhead. Consistent Hashing works by hashing every ID and then placing this hashed number in a 'Circular' key-space from 0 to 2^N.

For example let us assume that there is a key-space of $2^5 - 1$, this gives us a ring from 0 to 31 as shown in Figure 2.3. Let us assume that there are three storage nodes Node0, Node1 and Node2 whose ID's hash to 3, 11 and 19 respectively. This places them around the ring as shown in the Figure. Node0 is responsible for storing items that hash into the key-space from 3 till 10, Node1 is from 11 till 18 and lastly Node2 is from 19 till 2.

[8] `http://en.wikipedia.org/wiki/Distributed_hash_table`

[9] Cassandra was used by Facebook's inbox

If there is a request for an item whose hash is equal to 16, then this request will be forwarded to Node1 as this request falls into its range. Consistent hashing is used because this technique allows for scalability. If there is a new node added and it is placed in-between Node1 and Node2, then only Node1 needs to offload some of its data to this new Node and then depending on the routing schemes, some or all of the routing tables need to be updated.
Replication is achieved by sending a copy of the data to another Node in a consistent way. One could simply pass a copy to the next node in the key-space, or one could modify the ID of the data so that its hash sends the data to another region in the key-space.

The disadvantage of this is that each node needs to store a table containing a list of all other nodes and this can become inefficient when there is a significant number of nodes. For this reason some DHT systems break up the node list into finger tables which are stored at the nodes as is the case with Chord[29]. The nodes then use these finger tables to store the addresses of successors to which the query should be sent. There is ultimately a trade off between the number of hops and the size of the finger tables. In high performance systems one would want to decrease the number of hops so as to respond to queries in the least amount of time.

2.3.2 Cassandra

The Apache Cassandra project[10] is a highly available elastic fault tolerant distributed key-value storage system. It makes use of the partitioning and replication techniques used by Amazon Dynamo[9] and the data model as used by Google's BigTable storage system[6]. Cassandra is a peer-to-peer storage system that uses consistent hashing, as used in Dynamo, to achieve scalability and to remove a single point of failure. According to the CAP theorem by Brewer[5], one can only pick two of Consistency, Availability, Partition toler-

[10] `http://cassandra.apache.org/`

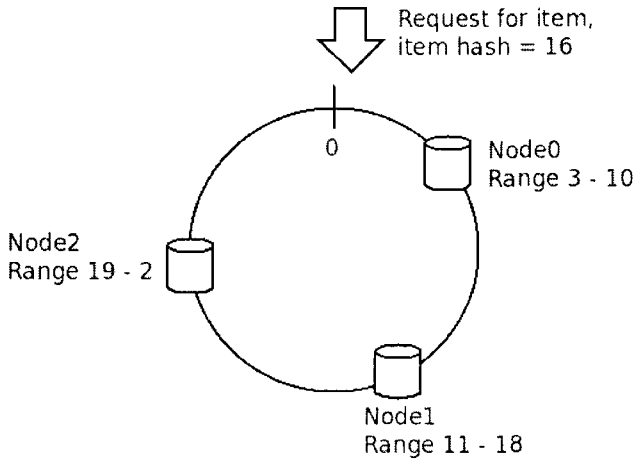


Figure 2.3: Consistent Hashing Example: key-space is 0 to 31.

ance. Cassandra has been designed to provide Availability and Partition tolerance with eventual consistency. When querying data, a client connects to a peer node which then services the request. If the peer does not have the information required by the client then the peer node requests the information from the correct node returning the results back to the client.

2.3.3 Hbase

The Apache HBase project is a distributed scalable Big Data Store[11]. It uses the column-orientated store modeled after Googles BigTable[6] and is now being phased into being used as Facebooks message store[4]. This design values Consistency and Partition Tolerance from Brewer's CAP theorem. With the HBase system, a client is responsible for locating the information. The client locates the region server the client directly, not through a master , and issues read/write requests directly.

[11] `http://hbase.apache.org/`

2.4 Secure Distributed Storage

As organizations store more and more data, the storage medium becomes a prime target for an attack by a malicious intruder[28]. The problem is that one cannot simply protect the perimeter. There have to be mechanisms in place that will protect the data even if there is a breach in the system.

One approach in doing this is to assume that the storage medium is already compromised. By assuming this the system can be designed in such a way so as to place no trust in the storage medium, so even if there is a breach then the data will remain secure. Another approach is to assume that all clients are unsecure, however this approach is not relevant to this work since the system being built in this paper will assume the the storage cloud is unsecure.

A survey performed by V.Ker et al.[20] examined a number of network systems and secure network files systems. In their survey they state that there are a number of requirements that need to be satisfied for a file system to be deemed secure. A secure storage system should provide Authentication and Authorization, Availability, Confidentiality and Integrity, Key Sharing and Key Management, Auditing and Intrusion Detection and lastly the system should have acceptable Usability, Manageability and Performance. The following systems are taken from their survey since the techniques used in securing the data could be beneficial in design a secure cloud based storage system.

2.4.1 Sirius

Sirius is a secure storage system developed at the University of Stanford by Goh et al. [12]. Sirius was developed to be a security mechanism that can be layered on top of any network file system such as NFS, CIFS, P2P.

Sirius runs a client side daemon that captures all network file system events and does the necessary security processing on data being sent to or received from the file system. Since all the security processing is done at the client it allows the data to be stored in an untrusted environment.

Each Sirius user maintains an asymmetric key for encryption and another for signing. These are called the *Master Encryption Key* (MEK) and *Master Signing Key* (MSK) respectively. File's are kept in two parts on the server. One contains that file's meta-data, access control information, and the other contains the encrypted file data. The file data is encrypted using a symmetric key called the *File Encryption Key* (FEK). The encrypted data is also signed using a *File Signature Key* (FSK).

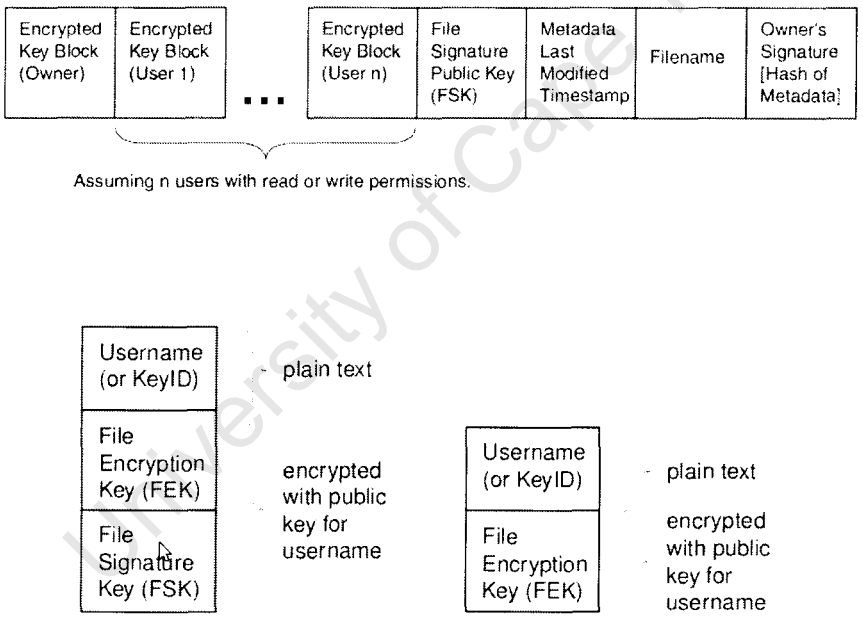


Figure 2.4: Sirius Meta data file

Figure 2.4 shows the structure of the Sirius meta-data file. Each *Encrypted Key Block* (EBK) is encrypted with the MEK. If a user has read rights then the EBK has the FEK, this allows the user to decrypt the data and read it. If a user has read/write rights then

the EBK has both the FEK and FSK. This will allow the user to modify and sign the data. Data can only be be modified if the data can be signed. Users can verify the integrity of the data by using the FSK to check the signed hash of the data.

This structure allows for file sharing since the owner of file can simply add an EBK for a new user with the appropriate permissions. Revoking a user means the file owner needs to generate a new FEK, the revoked users EBK is removed and the remaining user EBK's are regenerated by encrypting the FEK with the Public component of the users MEK.

2.4.2 Strong Security for Network-Attached Storage

Another system that relies heavily on client side security is Secure Network-Attached Disks (SNAD)[23]. From their observations, Miller et al. found that in the worst case their system imposed a penalty of 20% for larger sequential transfers and almost no penalty for random access.

Their design is to encrypt all data at the client and provide the server with sufficient authentication information. SNAD encrypts files at block level, this allows for efficient random access. Each file has a *key object*, the key object stores a signed hash of the file allowing for users to the check integrity. The key object also stores a list of tuples within the body of the key object. Each tuple has a UserID, Encrypted Password and the Permissions. The encrypted password is the File Encryption Key, encrypted with the users Public Key. The permissions field is used by the disk to determine whether the user has the rights to perform the action. Figure 4 shows the relationships between the objects within the system
Each SNAD node also stores a certificate object that holds user information such as the public key and an HMAC key.

When encrypting a file, SNAD encrypts each block and computes a hash over the entire

data object and then signs the hash with the users private key. The disk is able to verify the hash by decrypting the hash with the users public key and then recomputing the hash of the data object. If the hashes match and the user has write permissions then the file is written do disk.

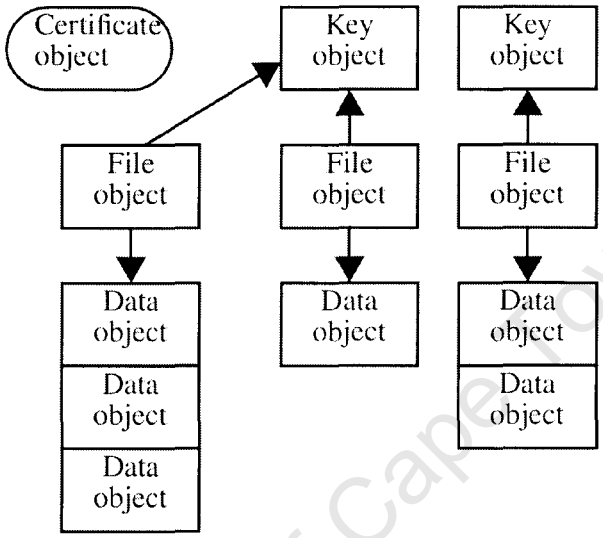


Figure 2.5: SNAD object relationships

2.4.3 Plutus

Plutus[16] is yet another system that secures data by performing client side encryption at block level and not trusting the storage system. The Plutus system groups files together that have the same user permissions. This technique lowers the amount of keys that need to be generated and maintained. Plutus implements read-write differentiation by using a similar technique used by Sirius[12]. A writer has both a file signing key and a file encryption key while the reader has only a file encryption key.

Plutus uses a technique call *key rotation* to revoke user rights. Key rotation creates new versions of keys whilst allowing readers to generate previous key versions. If a user is

revoked, then a new generation of the key is created and all valid users are given this key. This allows revoked user to still read files that were accessible at the time of key revocation, however any subsequent changes are unreadable by the revoked user. An example of this is illustrated in Figure 5

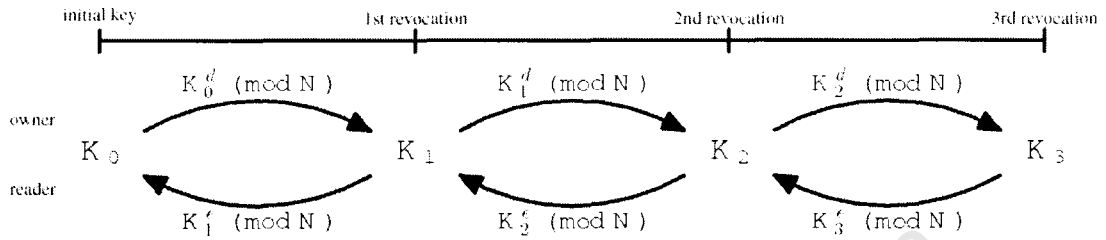


Figure 2.6: Illustration of the Plutus key rotation scheme. The owner uses (d,N), the private key, to generate new keys. The readers use (e,N), the public key, to generate older versions.

2.5 Internet Scale Communication Protcols

2.5.1 REST Interface

The REST protocol is the idealized model of interactions within a web-application. It has been designed to meet the needs of Internet-Scale distributed hypermedia systems by allowing for scalability, general interfaces and independent deployment[15]. The most widely used protocol for REST is HTTP. HTTP defines a set of methods such as GET, PUT, POST, DELETE. These methods are used to access and manipulate resources over the Internet.

The REST architecture defines three elements namely *Data Elements*, *Connecting Elements* and *Processing Elements*. Data elements are essentially resources that are exposed via REST. A resource can be anything that can be named. A Connecting element manage

the network connections. Connecting elements present a general abstract interface for communication. A Processing element is essentially any component within the web-application that handles requests and processing[15].

REST places restrictions onto component interactions. It ignores implementation details to achieve scalability, independent deployment and allowing intermediary components to reduce interaction latency.
An example of a REST call is shown below

```
http://www.acme.com/phonebook/UserDetails/12345
```

2.5.2 Simple Object Access Protocol (SOAP)

SOAP, like REST, is used for exchanging information in a web services environment. It uses XML as its message format and usually relies on Remote Procedure Calls or HTTP for transmission. A SOAP message can be sent to web-service-enabled web site with the necessary parameters for a certain function.[12] This communications protocol is used by the jets3t library that will be used in the implementation of the prototype[13] for this study.
An example of the a call via SOAP is shown below, it equivilent to the call shown in the REST example

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:body pb="http://www.acme.com/phonebook">
  <pb:GetUserDetails>
```

[12] http://en.wikipedia.org/wiki/SOAP
[13] http://jets3t.s3.amazonaws.com/toolkit/toolkit.html

```
    <pb:UserID>12345</pb:UserID>
   </pb:GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

2.6 Searchable Encryption

The idea of searchable encryption is to allow an untrusted server to perform some operation on the encrypted data without gaining any knowledge of the information within. Boneh et al. [2] describe this as Alice having a number of devices: Laptop, desktop, pager, etc. The email gateway that Alice uses is supposed to route emails to the appropriate device based on keywords within the email. The problem is that Alice does not trust the email gateway thus all emails are encrypted. In [2], the authors describe a technique for allowing the gateway to route emails to specific devices based on these encrypted keywords without gaining any knowledge of the contents.

Such techniques can be applied to cloud storage by having the cloud service provider return query results based on the keywords attached to the data. If there is a corporation that has a number of encrypted documents stored on the cloud and a user wishes to retrieve all documents with the keywords "Project XYZ, John Smith". The cloud provider can then use the techniques described in[2] process the query in such a manner that the cloud provider is unable to infer any information about the user data.

Boneh et al.[2] developed a scheme that does the above by using bi-linear maps. The sender encrypts the message and keywords using the receiver's public key. The receiver then generates *trap doors* that will be used by the gateway to determine whether the message has the keywords.

Both [7, 27] describe various schemes that can be used to apply searchable encryption to documents such as using indexes to map keywords to encrypted documents. Such techniques can also be used within a cloud storage setting to make searching an retrieval more efficient. The system described in [27] makes use of stream ciphers to encrypt the data while the scheme described in [7] uses a pseudo random number which is then XOR-ed together with the encrypted words to provide searchable encryption. The work done by Curtola et al.[8] uses symmetric encryption to provide encryption with keyword search. This can be used to provide more efficiency as symmetric encryption is much faster than asymmetric encryption schemes. The only problems are that such schemes do not work well when sharing with multiple users. This does not fit well with the Cloud Storage setting as such a system would need to be able to support multiple users. There could be a way of merging the two techniques to gain the multiple user support of asymmetric encryption and have the better performance of symmetric encryption.

The work of Waters et al[30] describes an implementation of an encrypted searchable audit log using both Symmetric and Asymmetric encryption. Their system allows users to submit encrypted search capabilities that are then processed by the audit log servers to return the log entries containing the keywords specified. The symmetric approach was used for this research and is explained further in Section 4.5. Their asymmetric scheme uses an Identity-Based encryption scheme of Boneh and Franklin in[3]. The advantages of using the asymmetric scheme is that there are no secret keys for an attacker to steal. This however is not an advantage for the research in this thesis since the secret keys are stored at the client.

2.7 Conclusion

In this chapter we have discussed the underlying principles that will be used in building a secure cloud storage system. We started by introducing the general concepts in Cloud Computing and Cloud Storage. We then discussed topics in distributed storage systems as well as some examples such as Cassandra and HBase. We then discuss how to secure distributed storage systems as well as the general method of communication via REST and SOAP interfaces. Lastly we introduced to concept of searching for keyword across encrypted data.

Chapter 3

Design Requirements

This chapter will address the needs of a secure storage system that should be fulfilled in order for it be deemed secure. In thinking about the requirements at abstract level, it is also important to think about the use cases of the system. There are a number of applications for such a system such as an *Emailing System*, an extension for *Dropbox*[1] or a *Music Storage Application*. Assume that this system would run as a service, accepting requests from other applications. An emailing system could send a request to this system specifying the files to be backed-up to S3, the keywords to be attached and any additional security operations. The use of Dropbox as an on-line storage service has gained popularity, this system could be adapted to provide secure searchable encryption for a Dropbox service. One could select the files to be uploaded then specify keywords and this system would perform all the encryption and other security operations before the files are pushed to the Dropbox service.

Another potential use could be to store music on-line where the keywords of the song could be the artist, album and genre.

3.1 Design Criteria

In order for a system to be secure it must satisfy a number of criteria namely V.Ker et al.[20] provide a survey of the common secure storage systems and created a list of what requirements are needed in order for a system to be secure. The following list is an adaptation of this list and will be used to guide the design developed in this research:

[1] `http://www.dropbox.com/`

- Ensure the *Confidentiality* of the data being stored
- Maintain the *Integrity* of the data to ensure that it has not been tampered with.
- Ensure that *File Sharing* can be catered for.
- Allow for *Key-Revocation* when user rights need to be removed.
- Ensure that there is *Searchability* with in the encrypted data.
- Ensue that the system can recover from a *Compromised Key-pair.*

3.1.1 Confidentiality

The system needs to maintain *Confidentiality* of the data being store on the Public cloud. This means that only authorized users should be allowed to read the contents of the file. The system should not rely on the underlying storage to handle this as it is assumed that the underlying storage system is untrusted.

3.1.2 Integrity

The system should ensure that tampered or corrupted data is detectable. This includes unauthorized modifications to data by users with only read rights. The system should be able to detect when such events occur.
There also needs to be a method to ensure that the access rights and other file meta-data have not been tampered with. This is to be modified by the file owner only, no other user should be allowed to change user rights.

3.1.3 File Sharing

File sharing is an important aspect in any work or personal situation. It is important for this system to allow file owners to grant read rights and read/write rights to other users in a secure way. Users with read permissions should have access to the "data encryption

keys" while users with write permissions should have some means of sign changes made to the data.

3.1.4 Key-Revocation

As file sharing is important so is the need to revoke user rights. If a user's rights are to be revoked it needs to be done in such a manner so that there is no way that a user can have further access to the confidential data. There are however two ways of revoking user keys, the one is to immediately revoke the key and the second is to revoke the key only when there is a change to the data. The logic behind this is that there is no loss of confidentiality to unaltered data, since the revoked user is not learning anything new.

3.1.5 Searchability

There have been advances in cryptography to allow for users to search through encrypted text as has been documented in Section 2.6. This can allow a storage system to receive a query, search for relevant matches and return the results without gaining any knowledge about the contents of the query or the results. This technique will allow users to safely store vast amounts of data on a public cloud and query for specific files without losing any confidentiality.

3.1.6 Compromised Key-pair

Should a users key-pair be compromised, there needs to be mechanisms in place that will allow for the system to recover from such a compromise. All files to which this user had rights to need to be re-keyed and re-encrypted.

3.2 Assumptions and Constraints

It is assumed that the owner of the file trusts the users to which access rights are granted. Trust in this context means that the user will not decrypt the data and distribute it. The user will not attempt to replicate the file under his/her name.

It is also assumed that no encryption or decryption is done within the cloud, the only processing that will occur in the compute cloud is forwarding requests to the storage cloud and the processing needed for to respond to search queries. All encryption is to be done on the client before being uploaded to a public cloud. By not having any encryption or decryption occurring in a VM instance within the cloud, such as an EC2 instance, there is no potential risk of the public cloud being able to access any of the information since they do have priviledged access[24]. The solution is constrained to the class of applications where this paradigm makes sense.

3.3 Assessment

These design requirements will be used to assess the design to ensure that it has fulfilled all the requirements. Chapter 6 will test and assess how effectively these requirements were fulfilled and their performance implications.

Chapter 4

System Design

The aim of this chapter is to explain how the system has been designed to fulfill the requirements set out in Chapter 3. The chapter starts by discussing the overall architecture of the system and how the *Client* and *Server* applications interact. The various operations performed by both the client and server are introduced. The data structures used by the system are then introduced and explained, namely the the *Secure File Object*, *Secure File Object Key Words* and the Network Message. We then discuss the Authentication Protocol between the client and server. The file system operations are then discussed and formally presented. Next we discuss the searching functionality of the system which includes the generation of keywords and how the server performs the encrypted searches. Lastly the users of the system and their access rights are discussed.

4.1 System Architecture

The system comprises of two applications. A client application that runs on a user's computer, and a server application that runs on the compute cloud. The responsibility of the server application is to respond to client requests as shown in Figure 4.1.

The client handles all security processing so as to remove all trust from the cloud provider. The client is responsible for encrypting the file that is to be stored, performing key management, user access management and communicating with the application running in the compute cloud.

A user will log in to the client application by providing a password that accesses that user's key-store. Once logged in the user specifies the file which is to be saved onto the cloud along

with performing any other administration tasks as discussed later. The client application then authenticates itself with the server, as explained in Section 4.3, creates a message and sends it securely to the server application running on the compute cloud, as explained in Section 4.4. This server application checks the message header to determine the task that needs to be completed.

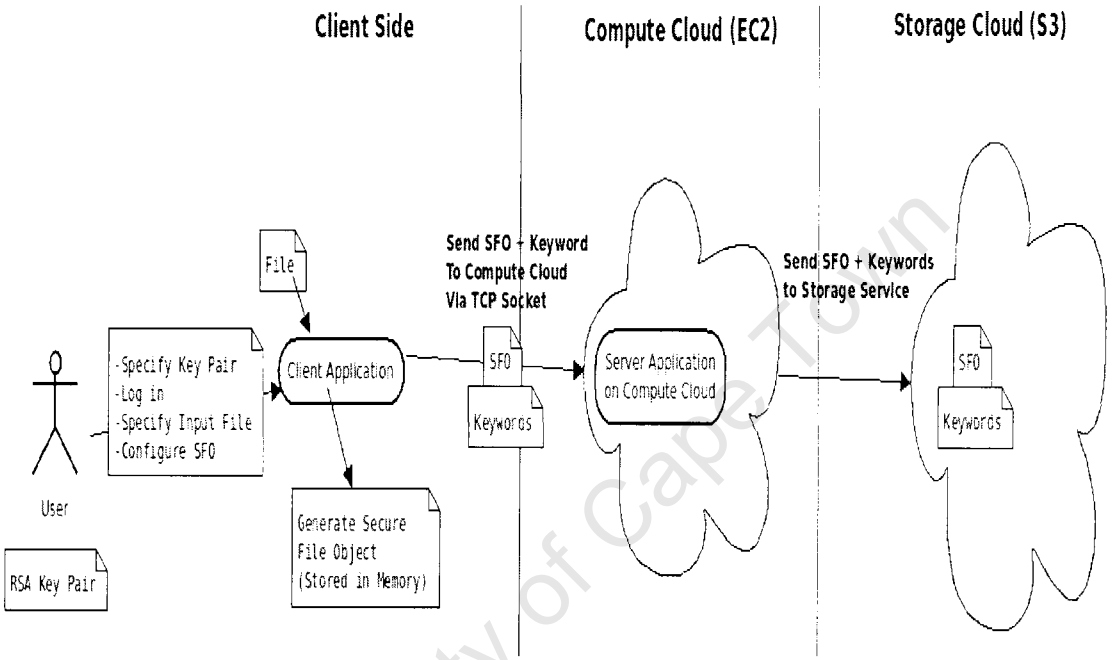


Figure 4.1: Overall system architecture

4.1.1 Client Application

The client performs all the security operations necessary to securely store data on an untrusted public cloud provider. Each user of the system is required to have an RSA key-pair that will be used to sign data changes and access the *File Encryption Keys*. The client application creates *Secure File Objects*, all security operations are performed on this object. The security operations include:

- Add Data

- Add Users
- Remove Users
- Hash and Sign Data
- Hash and Sign Secure File Object
- Check Integrity.

The details of this object structure are discussed in Section 4.2.1.

Once the user has performed all desired operations on this object, the client then sends this object to the server application running on the compute cloud. The client can send a number of requests to the server namely:

- Put - Upload a file to the Storage Cloud.
- Get - Downloads a file from the Storage Cloud.
- Delete - Deletes a given object from the Storage Cloud.
- Ls - lists all the items within the Storage Cloud.[1]
- Search - This is used to search for a given keyword within the Storage cloud.

A typical flow of events would be for the user to log in, create a Secure File Object(SFO), place the contents of a file into the SFO, hash and sign it and send it off to the Cloud Provider. At a later stage, download the file from the storage service, check the integrity of the file and then possibly save the files un-encrypted contents to the local machine for use later.

[1]In S3 this will list all objects within a given bucket

4.1.2 Server Application

The server application is used as a lightweight interface between the client and the Cloud Storage Service. As discussed in Section 4.1.1, the server has to handle a few types of operations. Put, Get, Delete and LS (list bucket, similar to the Linux command) are all lightweight operations and merely generate a message that is then sent to and handled by the storage cloud.

The server handles the keyword search functionality of the system. The client sends a search capability which the server then uses to find all matching file names and return this list to the client, as explained in detail in Section 4.5.

4.2 System Data-Structures

4.2.1 Secure File Object

The *Secure File Object* is the container that is used to store data securely on the cloud. All security operations on this container are performed at the client, the server performs no modifications to this object. The structure of the secure file object is shown in Figure 4.2.

Each secure file object has a unique file encryption key, any symmetric encryption will suffice. The attributes of a secure file object are:

- File Name: The name of the object file.
- OwnerID : The ID of the owner/creator of the file.
- Last Modified User ID : The ID of the user that last modified the data.
- Read List : List of users with read access rights, mapping UserID to $[FEK]_{Pu}$
- Public Key List :List of users Public keys and an is-writer flag.

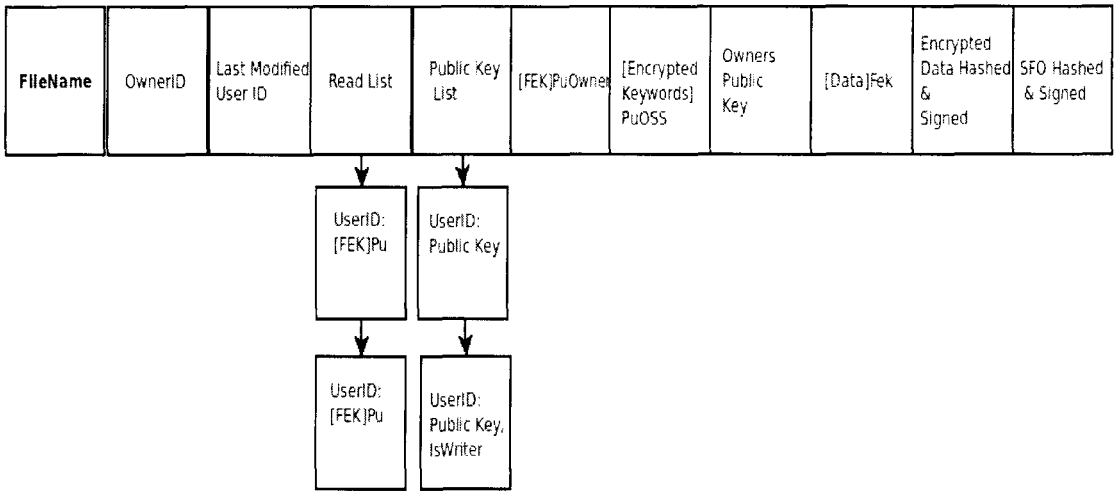


Figure 4.2: The Structure of the Secure File Object

- Encrypted File Encryption Key: The File encryption key is encrypted with the owners Public key.
- Encrypted Keywords: An encrypted string of keywords. This field is merely used administration purposes since the searchable keyword generation is a one-way function. It is encrypted with the same Symmetric Key that is used by the owner to create encrypted keywords by the owner, thus only the owner has access to this string.
- Encrypted Data Digital Signature.
- Secure File Object Digital Signature.

As mentioned in Section 3.1, there are a number of requirements that need to be satisfied by this system in order to provide a secure means of storage.

Confidentiality

The secure file object provides *Confidentiality* by encrypting the data field with a File Encryption Key. Only users with read(and write) permissions may read the contents of the

data field. The data field is encrypted using a Symmetric Encryption Algorithm, like DES. The File Encryption Key is only generated by the File Owner upon Secure File Object creation and when user rights are revoked.

Integrity

Integrity is maintained in two ways, firstly the integrity of the data field needs to be checked and secondly the integrity of the secure file object as a whole needs to be checked.

Data integrity is maintained by hashing the encrypted data field then signing this hash with the users private key. The user then sets the "Last Modified UserID" to his own user ID. Subsequent users look up the Last Modified Users Public Key in Public Key List, decrypt the signature and check the integrity of the encrypted data.
Secure File Object integrity is maintained by hashing all the fields of the secure file object except for those that users may alter(Last Modified UserID, Encrypted Data and Encrypted Data Digital Signature) and then encrypting the hash with the owners private key. The resulting hash is as follows:

$$\text{SFODigest} = [\text{F}(\text{FileName} \,|\text{OwnerID} \,|\text{ReadList} \,|\text{PublicKeyList} \,|[\text{FEK}]_{\text{PuOwner}} \,|\text{PuOwner})]_{\text{PrOwner}}$$

where F is a hash function such as SHA or MD5. The hash is then signed with the owners Private Key.
When a user loads the SFO from the cloud, the user then uses the Public key of the owner to decrypt the Digest and check the hash.

File Sharing

Like any file system, there needs be a way of sharing files with other users of the system, for this reason the secure file object makes use of two lists. The Read List is a list of user

id's $\rightarrow[\text{FEK}]_{\text{PuUser}}$. Since the FEK is encrypted with the users Public key, that user can gain access the FEK by decrypting it with his Private Key.

The *Public key list* maps user id's $\rightarrow$(Public Key, Is Writer) and is needed for two reasons, firstly it is used to maintain a list of all user public keys to be used in key revocation as explained in Section 4.2.1 and the secondly it is used to maintain a list of which users have write access to the data with the is-writer flag.

When a user downloads a SFO from the cloud, that user will use the "Last Modified UserID" to look up the public key of that user in this list and check whether that user does in fact have write access to this file.

Key-Revocation

With allowing access to users there need to be mechanisms in place for removing access to users. Revoking a users read access rights means that the owner needs to generate a new File Encryption Key (FEK) and re-encrypting the encrypted data field with a new FEK. Then the owner needs to remove the revoked user from the read list and the public key list. The owner then iterates through the public key list encrypting the new FEK with each users Public Key. The user do not need even know that a new FEK has been issued, since the whole process is transparent to them.

Revoking a users write access rights is to simply set the Is-Writer flag to false in the Public Key List. User revocation can only be performed by the owner. Once the changes have been made the SFO is signed again to maintain integrity. Since the SFO needs to be signed, only the owner can make these changes. This stops users from falsely assigning themselves write access or read access.

Malicious Users

The design of the system assumes that users can be trusted, i.e. if an owner grants access to a user, that user will not be malicious. However measures should be in place so the the system can detect any malicious activity.

If a user is granted read access, that user can modify the contents, generate a Encrypted Data Digest and set the Last Modified User ID field. However, when the next user loads this file, that user will do an integrity check. As mentioned an integrity check is done by looking up the Public Key of the Last Modified User; since this Is-Writer flag is not set, the system will generate an error because an unauthorized user has modified the contents.

If a read user adds write permissions then this change will be detected by subsequent users since the malicious user is unable to generate a *Secure File Object Digest*. This will inform subsequent users that there has been an unauthorized change to the secure file object.

There is another problem with malicious users that is not easily fixed and a suitable solution is beyond the scope of this research. That is the problem of a user taking ownership away from the file owner.
Since all read users have access to all fields of the SFO, they can create a new SFO (thus becoming the file owner), copy the encrypted contents of the old SFO along with the read and public key list, name the new SFO to that of the old SFO, and delete the old version. To the other readers the file will still look the same, however, the file has changed owners. A solution to this problem is to implement a Public Key Infrastructure (PKI) that allows users to check whether a File ID maps to a given Owner Public Key. As this is outside the scope of this research, a more naive approach was taken. The file owner stores a local copy of the file list. This file list is then used to do a reconciliation with the files store on the Storage Service. Should there be a difference, then the owner knows that some malicious

activity has occurred.

4.2.2 Secure File Object Key Words

The *Secure File Object Key Words*(SFO Keywords) is used to store the secure keywords as generated by the owner. For every SFO stored in the cloud storage service, there is an attached SFO keywords File. These files are used by the server application to perform the cryptographic operations necessary for searchable cryptography as detailed in Section 4.5.

The SFO Keywords three fields are as follows:

- Random Bit String.
- Flag.
- List of encrypted Keywords.

The use of these three fields in finding specific keywords is explained in more detail in Section 4.5.

4.2.3 Network Messages

This system uses two types of network messages: a *Request Message* and an *Authentication Token.*

The authentication token is used by the client to securely transmit data across the network to the server. The fields in the authentication token are as follows:

- Key / Container
- Encrypted Nonce

- Hash

These fields are used by the system in the authentication process to establish a session key. Key / Container field is used to transport the encryption keys as well as other information such as the access key hash. The encrypted nonce is used for freshness as well as authenticating the server. The hash field is a digest of the message, the details of which will be explained in Section 4.3.

The *Request Message* is used for the file-system operations of the system. This message is sent from the client to the server instructing the server which operation to perform. The fields of this message are as follows:

- Container
- Message Type
- Digest

The Container field is used to pass any additional information along with the request, such as a *Search Capability* or an *Object ID*. The Message Type field instructs the server as to the type of request that is being sent and the Digest is used to maintain Integrity. The details of these operations and the use of the Request Message will be explained in Section 4.4.

4.3 Authentication Protocol

Each client has a local copy of the server's public key. The client uses this key to establish a secure connection between itself and the server. The client generates a symmetric session key, encrypts this key with the server public key and sends it to the server. The client then sends its' public key, a nonce and the *Access Key* hash all encrypted with the session key to the sever.

In order for the server to connect to the storage service it needs these Access Keys[2]. The server can then validate the user by hashing its Access keys to those received from the client. If the hashes match, then the client is allowed to access the system. The server then stores the client's public key and sends the incremented nonce back to the client. If the client receives the correctly incremented nonce the client knows that it is securely communicating with a valid server. The protocol is displayed formally as follows.

Using the following definitions:

C: Client

S: Server

K_S: Session Key

C_{Pu}: Client Public Key

C_{Pr}: Client Private Key

S_{Pu}: Server Public Key

S_{Pr}: Server Private Key

AK: Access Keys

The protocol can be represented as follows:

C $\rightarrow$ S: [H[SecretKey |AccessKey]]E_{Ks}

C $\rightarrow$ S: [[CPu]E_{Ks} |[Nonce]E_{Ks} |[H[AK]]E_{Ks}]

S $\rightarrow$ C: [Nonce+1]E_{Ks}

After establishing a secure connection with the server the client is then able to securely transmit messages with the server using the session key, the details of which will be explained further in Section 4.4.

[2]In the case of S3 a secret key and access key is needed

4.4 File-System Operations

Once a secure connection has been established between the client and the server as explained in the Section 4.3, the client can then send file system operation requests to the server. If the client wishes to send an object to the server, the client creates a new secure file object and then sends this object to the server along with a digest encrypted with the session key. The server can thus confirm the integrity of the message and the authenticity of the object, formally shown below.

Using the following definitions:

C: Client

S: Server

K_S: Session Key

MT: Message Type (File System Operation)

Co: Container

SH: SFO Hash

SFO: Secure File Object

The protocol can be represented as follows:

4.4.1 Upload Commands

Such as sending a file to the Server

C $\rightarrow$ S: [MT |Co |[H[MT |Co]]E_{Ks}]

C $\rightarrow$ S: [SFO]

C $\rightarrow$ S: [MT |SH |[H[MT |SH]]E_{Ks}]

4.4.2 Download Commands

Such a receiving a file or file list from the server.

C $\rightarrow$ S: [MT |Co |[H[MT |Co]]E_{Ks}]

S $\rightarrow$ C: [SFO]

S $\rightarrow$ C: [MT |SH |[H[MT |SH]]E_{Ks}]

4.5 Searchability

This is the operation when users can submit queries to the server which them returns results back. The server performs cryptographic operations over the files in the storage service, returning the results of the query, where there is a match. The scheme used by the server to determine the results of the query is derived from the Searchable Symmetric scheme that was developed in [30]. Waters et al. [30] developed two schemes in their research; a symmetric and an asymmetric searchable scheme. The symmetric scheme was chosen as it has a performance advantage over the asymmetric scheme. The advantages of an asymmetric scheme over the symmetric scheme are also not applicable to our setting. It is important to note that generating these searchable keywords is a one-way process. Once a searchable keyword is generated, there is no way of retrieving the original keyword. For this reason there is the encrypted keywords field attached to each SFO. So that the owner may check which keywords have been attached. This extra field in the SFO plays no part in the query process though.

4.5.1 Secure Keyword Generation

In order to provide the secure searchability functionality, there are number of extra parameters needed. Each list of keywords needs a **flag** and a **random bit string** r as stored

in the Secure File Object Keywords file Section 4.2.2. Let Wi be the i-th word in the keyword list. H_S is a hash function keyed with the secret S. Hmac-SHA1 is used for the hash function H and *padding* is some random bits. For each keyword the server computes the following:

$$a_i = H_S(Wi),\ b_i = H_{ai}(r),\ c_i = b_i \oplus (\text{flag} \mid \text{padding})$$

c_i is the encrypted keyword that will be used by the server to determine which c_i matches a given search capability. The list generate by creating c_i from each w_i is stored together with the SFO when uploaded to the cloud storage service.

4.5.2 Keyword Search

Once the owner has created an *Encrypted Keywords List* (EKL) for an SFO and uploaded this SFO along with the keywords to the cloud storage service, users are able to submit *search capabilities* and receive search results.
Search capabilities can only be generated by the file owner since the secret key S is needed. If a user wishes to get a search capability for keyword W then the owner will generate it as follows:

$$d_w = H_s(w)$$

The Owner then provides this search capability to the user, who then sends it to the server. Upon receiving d_w the server then performs the following operations per file.

$p = H_{d_w}(r)$, since r is stored with the encrypted keyword list. For each c_i in the EKL the server computes:
$x = p \oplus c_i$.

If the first l bits of x match the *flag* then there is a match for the current file. It returns the file name as one of the search results. This operation is repeated for every file in the Storage Service.

4.6 System Users

As is with most file systems, there are different types of users with different capabilities. In the case of this system there are three different types of users namely:

- Owner: File Owner.
- Writer: User with Write access, read access is implied.
- Reader: User with Read access.

Figure 4.3 shows the use case diagram for all users. This diagram illustrates all the actions that can be performed by all the users of the system.
As shown in the figure, all users can Load a file from the cloud, assuming they have access to the access keys. All users can also check the integrity of the file, list the users and rights assigned to a specific file. Save the SFO back to the cloud and output the contents of the SFO to a file or terminal.

Users with read access have all the rights shown in Figure 4.3 along with the ability to read the contents of the encrypted data field.

A user with write access means that such a user can perform the actions that 'All' users can along with the rights shown in Figure 4.4. The write user can edit the encrypted data of the SFO and sign the changes.

The File Owner can perform any action on a Secure File Object, the owners rights are that of a write user, read user and all user rights along with the rights shown in Figure

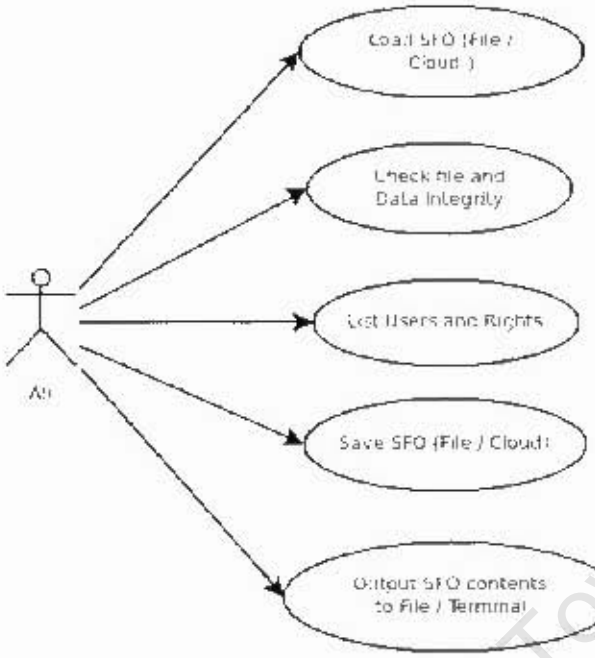


Figure 4.3: Use case for all users

4.5. As shown in Figure 4.5, the owner can create an SFO, place contents into the SFO (contents mean the data placed into the encrypted data field). The file owner can add and remove users, sign the encrypted data field and sign the SFO. The owner and only the owner has rights to attach keywords to the SFO and generate search capabilities which can then be used by other users to search for files containing those keywords.

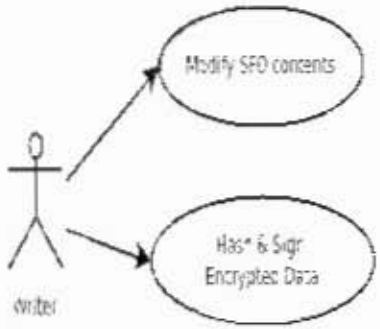


Figure 4.4: Use case for Write users

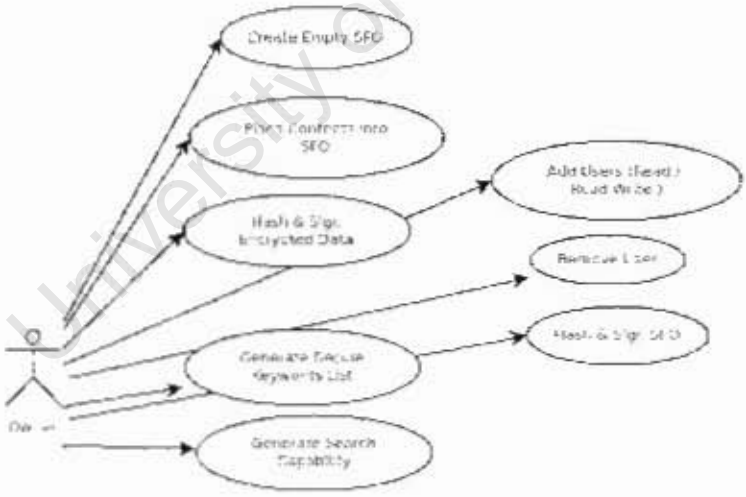


Figure 4.5: Use case for the File Owner

4.7 Conclusion

This chapter has described the architecture, data structures and algorithms needed in order to satisfy the requirements specified in Chapter 3. The chapter discussed the client / server architecture and responsibilities of each. Then the data structures used by the system are explained in detail, namely the Secure File Object, Secure File Object keywords and the Network Messages. The chapter then explained how clients authenticate with the server and how the file system operations are executed securely. The use of searchable encryption is then explained, showing the algorithms needed in order to provide such functionality and lastly the users of the system and their various rights were explained.

Chapter 5

Implementation

A prototype comprising of a client application, server application with a communication interface for Amazons S3 was implemented using Java to run on a Linux operating system and uses standard TCP sockets for communication between the client and server. The server runs on an EC2 Micro Instance and uses the JetS3t[1] library to connect and communicate with Amazon's S3. This chapter begins by explaining some of the implementation details of the client and server applications along with the algorithms and procedures needed to satisfy the requirements.

5.1 Client Application

The client application is used by users to connect and transmit messages with the Server running on EC2. Each user has a 1024 bit RSA Key-pair that is stored in a key-store file. Each file owner has Searchable Key Store file that is used to generate search capabilities as described in Section 4.5. This file stores two keys, one is the SecureFileObject keyword key which is used to symmetrically encrypt the list of keywords attached to an SFO so that they may be later read. This key is a standard DES key. The next key is an HMAC-Sha1 and is used to generate searchable encrypted keyword lists that will be used by the server to respond to user queries. The system uses the Java KeyGenerator class to generate the specific keys.

The client application also contains a settings file that is needed to specify the location of the users key-store file, the UserID, the EC2 secret and access keys and the address of the

[1] `http://jets3t.s3.amazonaws.com/toolkit/toolkit.html`

EC2 instance to which the client must connect.

Communication between the client and the server is achieved by serializing the Network Message, Secure File Object and Secure File Object Keywords data structures and sending those over a TCP socket using the Java ObjectStreamWriter or ObjectStreamReader.

5.1.1 Client Algorithms and Procedures

The following section will explain some of the core procedures of the client application and how the client system was broken up in to various components.

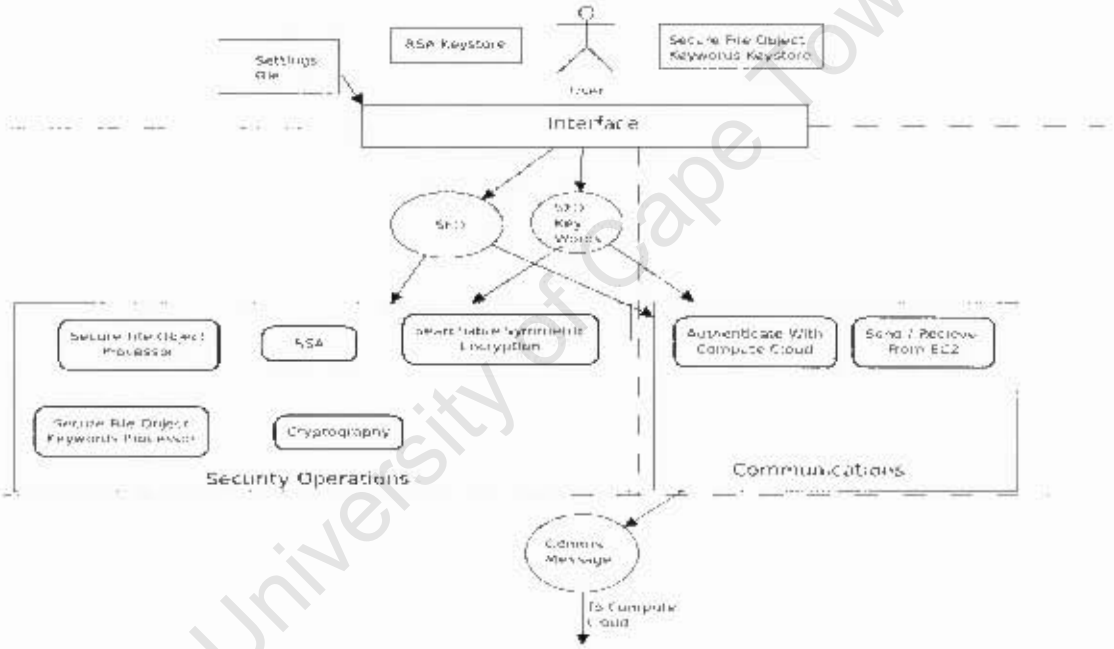


Figure 5.1: System Components

Figure 5.1 shows the various components and classes that are used by the client application. The system can be broken up into two major components; the Security Component and the Communications Component. The interface is used by the user to interact with the system. Upon starting the application the Settings file is read by the application and

the RSA Key-store along with the Secure File Objects Key-store are loaded. The RSA class is used to extract the Public and Private keys from the RSA Key-store, similarly the Secure File Keywords Key-store class is used to extract the encryption keys from the Secure File Objects Key-store file. The Secure File Object Processor is used by the system to perform various security operations on a Secure File Object. The Cryptography class is used to preform cryptographic operations on data, this includes encrypting a byte array using asymmetric encryption or symmetric encryption and generating message digests from byte arrays.

Security Operations

The Secure File Object Processor class, as is shown in Appendix A.2, is used to perform Cryptographic operations on an SFO instance, the implementation of which is shown in Appendix A.1. It makes use of the cryptographic operations exposed by the Cryptography class. The SFO Processor provides a set of high level operations to the interface such as shown in Table 5.1

The *Create SFO Object* command creates an SFO object in memory. This function takes the file's owner's UserID and RSA key-pair. This function creates the File Encryption Key and encrypts it using the owner's public key.

The *Add Read User* command adds a new entry to the *Read List*. This function takes the user's ID, user's public key and the owner's private key, encrypts the file encryption key with the user's public key. The function also adds a new entry to the *Public Key list* with the *Is-Writer* flag set to False. It then creates a digest of the object and signs it with the owner's private key.

Table 5.1: Secure File Object Processor Functionality

Create SFO Object	Add Read User
Add Write User	Remove Read User
Remove Write User	Hash Secure File Object
Hash Encrypted Data	Sign Digests
Verify Secure File Object Integrity	Set Data From File
Get Data	Save Data to File
List User Rights	

All block encryption was performed using the DES encryption algorithm [2] , with a default Java, key size of 64 bits as specified in FIPS PUB 46-2 [3]. The creation of hashes for both the data fields and the secure file object was performed using the MD5 algorithm.

Keyword Generation

As explained in Section 4.5, a random number R and a flag F are needed to generate encrypted keywords, the class definition of the Secure File Object keywords is shown in Appendix A.4. R is a 512 bit random number and F is a 128 bit random number seeded with the system time. The system receives a number of space separated keywords which are then processed as explained in Section 4.5 and stored in a list of byte arrays. These operations are performed by the Secure File Object Processor shown in Appendix A.5. The Secure File Object Processor uses the Symmetric Searchable Encryption class, Appendix A.6, to convert the keywords to searchable keywords using the algorithm specified

[2] Although AES of any other block cipher could also be used

[3] `http://www.itl.nist.gov/fipspubs/fip46-2.htm`

in the Section 4.5.1.

Communications Operations

Communication with the compute cloud is a two step process. The client first authenticates with the server, then exchanges data.
Authentication is explained in Section 4.3, once authenticated the client can then send *Network message* objects to the server to instruct it what operations to perform. This is achieved by creating a Network Message object and setting the *Message Type* field and possibly sending additional information that is placed in the *Container Field*.
For example, if the user wishes to *Put* an object, the client will set the message type field to PUT and send that object, then send the Secure File Object and potentially the Secure File Object Keywords to the server. Should the user not have specified Keywords for a Secure File Object, then client sends a NULL value for the Secure File Object Keywords. If a user wishes to *Get* a file, then client will set the *Message Type* field to GET and set the *Container* to the Object-ID that the user has requested.

Client Interface

The client application needs to provide the ability for other applications to interface with it. As mentioned in the use cases in Section 3, an application running on a users computer would need an interface to connect with and send instructions. This functionality was implemented with the use of sockets. The implementation has a thread running and waiting for connections from applications. The application sends a list of commands and parameters across the socket for the client which the client then processes. As an example, the application could send the following string. "-password=pass -create=secureTextFile -add=/home/Bob/TextFile -keywords=Bob TextFile -sign -put". That command passes through the key-store password, the name of the SFO instance, the location of the contents, the keywords that are to be attached, then a sign instruction which creates both the encrypted data hash and the SFO hash and lastly there is a put command which instructs

the client application to upload the SFO to the cloud. This approach allows any application to simply open a socket, connect to the client, pass through some commands which the client then processes and passes them onto the EC2 server. The process is illustrated in Figure 5.2.

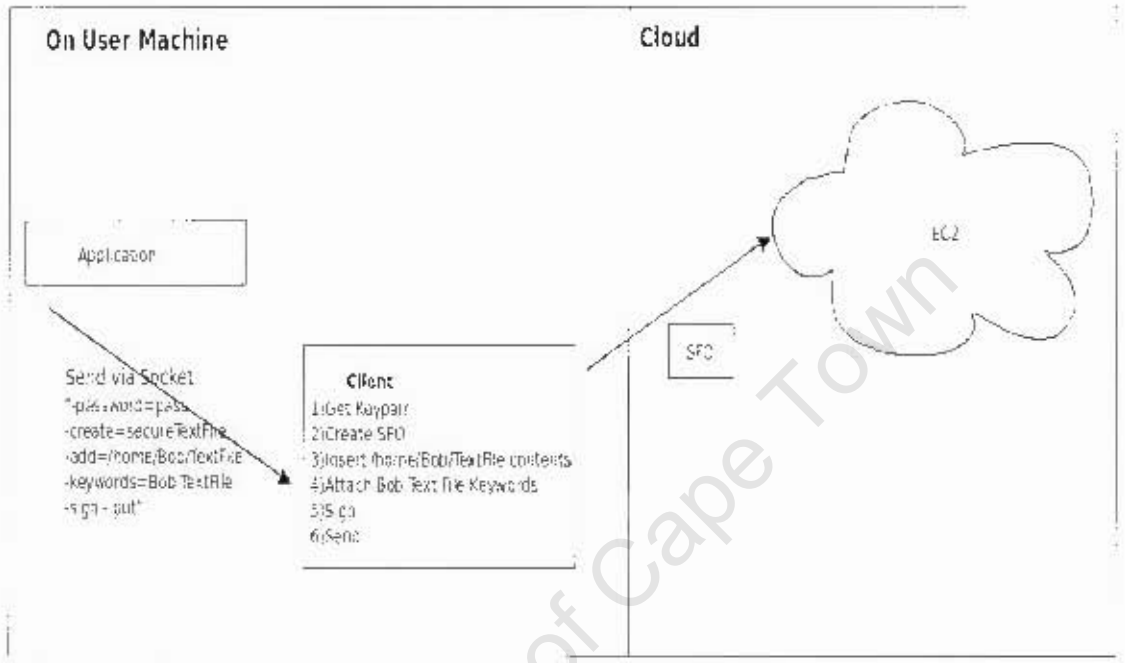


Figure 5.2: An overview of the Client Interface for Applications. Illustrates how an application can connect to the Client via sockets and send instructions.

5.1.2 Server Algorithms and Procedures

The server application runs on an EC2 Instance and waits for connections from clients. When a client connects, it is first authenticated then the server can receive instructions. Most of the instructions are simply pushed forward to the cloud storage service API[4] via our AmazonsS3 abstraction class which is defined in Appendix A.3. When a server receives an SFO Object, along with its keywords object. It serializes these objects to S3 using

[4]JetS3t

the name specified in the SFO, and for the keywords file it appends 'keywords.' to the beginning of the name.

When the server needs to search, it will receive a *Network Message* with the *Message Type* field set to *Search* and the search capability stored in the *Container* field.

The server then requests a list of all the files starting with 'keywords.'. Then for each file in the list, the server extracts the keywords in that file, and compares them to the search capability as explained in Section 4.5.2 and shown in Figure 5.3.

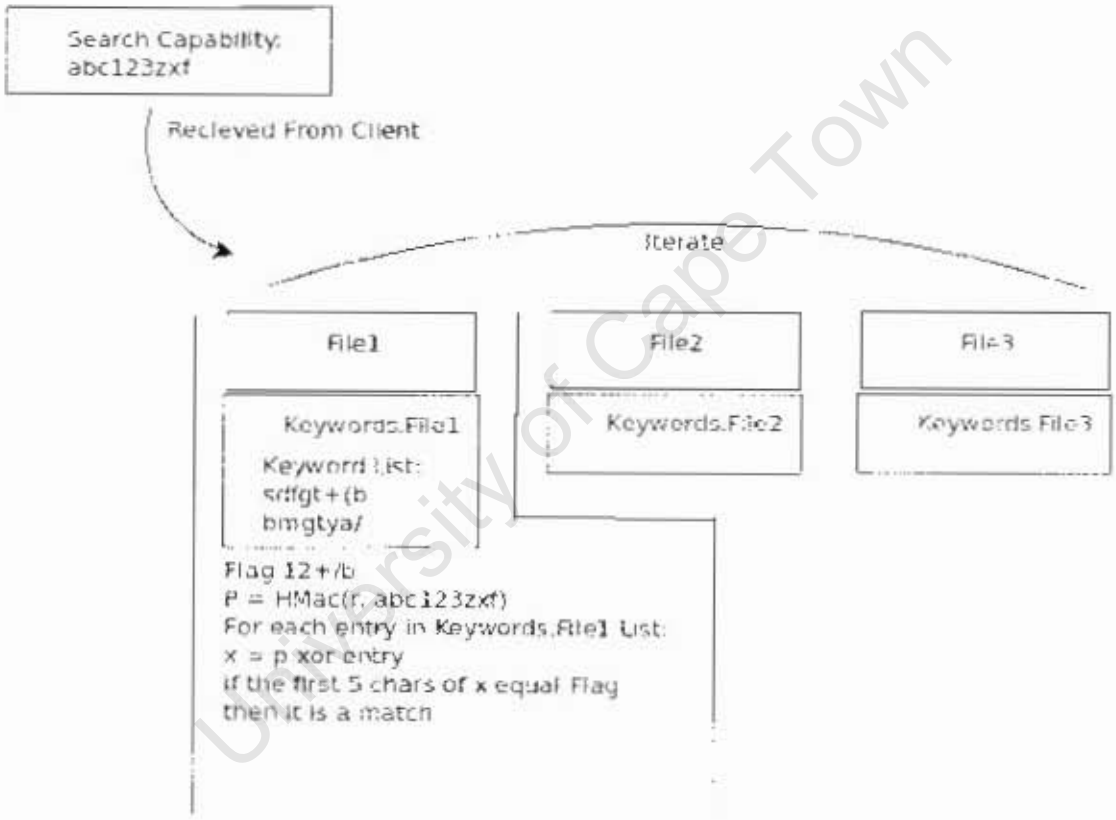


Figure 5.3: Search Implementation Example

Figure 5.3 shows an example of how the searching algorithm works. Assume there are three files stored on S3 each with a list of keywords. If the server receives a capability 'abc123zxf' then the server computes P = HMac(r,abc123zxf) for each keyword file. For

example assume that there are two encrypted keywords stored in 'keywords.file1' namely; sdfgt+(b and bmgtya/. The server will compute x = P xor sdfgt+(b and x = P xor bmgtya/, if one of the x's starts with the flag bmgtya then there is a match and 'file1' has the given keyword attached to it. This process is repeated for 'keyword.file2' and 'keyword.file3'.

5.2 Jets3t

Jets3t is a library that is provided by Amazon.com to provide developers with an easier way of connecting to the Amazon Simple Storage Service. In our implementation we abstracted away certain operations to make it simpler to upload files through the JetS3t library. We implemented the PutSecureFileObject method which takes a Secure File Object as a parameter and uploads it to S3 via the JetS3t library, all object casting is handled within in this method. Similarly we implemented the GetSecureFileObject method which takes an object key as a parameter and returns a SecureFileObject or NULL if the object key was not found in the S3 bucket. The abstractions that we implemented can be seen in Appendix A.3.

5.3 Conclusion

This chapter described how the specifications described in the Chapter 4 were implemented. We provided further details about the implementation of both the client and the server. We discussed the implementation of the algorithms and procedures, as well as described the different components that were implemented in accordance with the requirement goals. We then discussed how the server was implemented and the searchable encryption functionality of the system as well as the different parameters needed. Lastly we discussed the JetS3t library and the abstractions we added to make it more usable by our system.

Chapter 6

Testing

The purpose of the testing chapter is to assess whether the implementation has addressed the requirements specified in Section 3 and measure the performance and storage overheads. To achieve this we performed a set of tests for each of the functions specified in the Requirements. We then measured how long it took to secure data, the storage overhead of securing the data, the time taken to upload data unsecurely versus securely, and lastly we measured the time taken to execute queries at both the client and the server.

6.1 Configuration

The client testing was performed on a Intel® Core™2 Duo CPU T8100 @ 2.10GHz with 2004Mb of Memory running the Ubuntu 10.04 release. The EC2 instance running the server application was a *Small Instance* which has 1.7GB of memory, running one *EC2 Compute Unit* on a 32-Bit platform with *Moderate* I/O Performance. The Amazon Machine Image used was *ami-339ca947* running in the *eu-west-1 region*.

6.2 Functional Testing

As explained in Section 3.1, there are a number of design criteria requirements that need to be satisfied for a secure file storage system. The aim is to test whether the design of the system can be implemented and that it in fact satisfies the requirements at a functional level. The functional testing results are then summarized in Table 6.1.

6.2.1 File Sharing

File sharing functionality means that the owner of the file is able to grant other users permissions to read or modify the contents of the data. In order to test this, a separate / stand-alone user key-pair was created for 'Bob' and the following steps were performed.

FS1

1. File Owner, Alice, creates new Secure File Object and inserts data.
2. Create a new user key pair for Bob.
3. Bob logs in and tries to access data of the new Secure File Object, however Bob is unable to access the data since Alice has not granted him read access rights.
4. Alice adds a read user entry for Bob.
5. Bob tries to access the data again and is now able to since he has access to he File Encryption Key.

Similarly we had to test the write access functionality, for this we used Bob's key-pair again with the following steps, it is also assumed that Bob already has read access rights. The following steps test whether Bob's illegal changes can be identified.

FS2

1. File Owner, Alice, creates new Secure File Object and inserts data.
2. Bob logs in and modifies the data, without having write permissions.
3. Bob then signs the changes with his private key and sets the last modified users value to his ID.
4. Alice logs in, checks the SFO integrity and then checks the last modified user is set to Bob.

5. Alice then looks up Bobs public key entry which does not have the write flag set.

The following test checks the "normal flow of events", where Bob has write access and legitimately changes the data.

FS3

1. File Owner, Alice, creates new Secure File Object and inserts data.
2. Alice grants Bob write permissions.
3. Bob logs in and modifies the data.
4. Bob then signs the changes with his private key and sets the last modified users value to his ID.
5. Alice logs in and checks the last modified user is set to Bob. Alice looks ups Bobs public key and uses this to decrypt the signed data hash and determine the integrity.

The tests have been performed to determine whether the file sharing functionality correctly shares data between users. From the tests we can deduce that our scheme is **Successful.**

6.2.2 Key-Revocation

When a user is no longer allowed to read the contents of a Secure File Object, that user must be removed from the read-list and new File Encryption Keys must be generated. The following steps test whether this was correctly fulfilled.

KR1

1. Alice creates a new Secure File Object and inserts data.
2. Alice grants Bob read permissions.
3. Bob logs in and can decrypt and read the data.
4. Alice then revokes Bob's read permissions.
5. Bob can no longer decrypt the contents of the data field.

From this test we can deduce that our scheme is **Successful** in providing Key-revocation functionality.

6.2.3 Compromised Key-pair

A compromised key-pair means that a user's key-pair cannot be trusted anymore. This implies that the data needs to be re-encrypted. To test this functionality we performed the following steps.

CK1

1. Alice creates a new Secure File Object and inserts data.
2. Alice grants Bob read permissions.
3. Bobs key-pair is deemed compromised thus a new key-pair is generated.
4. Revoke Bobs access rights. By doing this step, a new File Encryption Key(FEK) is generated and the data is re-encrypted. The users with read permissions, except Bob, are then given the new FEK by encrypting the FEK with their Public Keys.
5. The last step is to re-add Bob as a read user with his new key-pair, this results in the same users having read permissions however there is a new FEK.

The test has shown that our scheme can **successfully** recover from a compromised key-pair.

6.2.4 Confidentiality

Confidentiality is essential for this system as the central idea is to prevent cloud providers or attackers from being able to access private user data. We tested this functionality with the following steps:

C1

1. Create Empty Secure File Object with block encryption disabled
2. Load contents
3. Serialize the object to a file
4. Open serialized file with a text editor

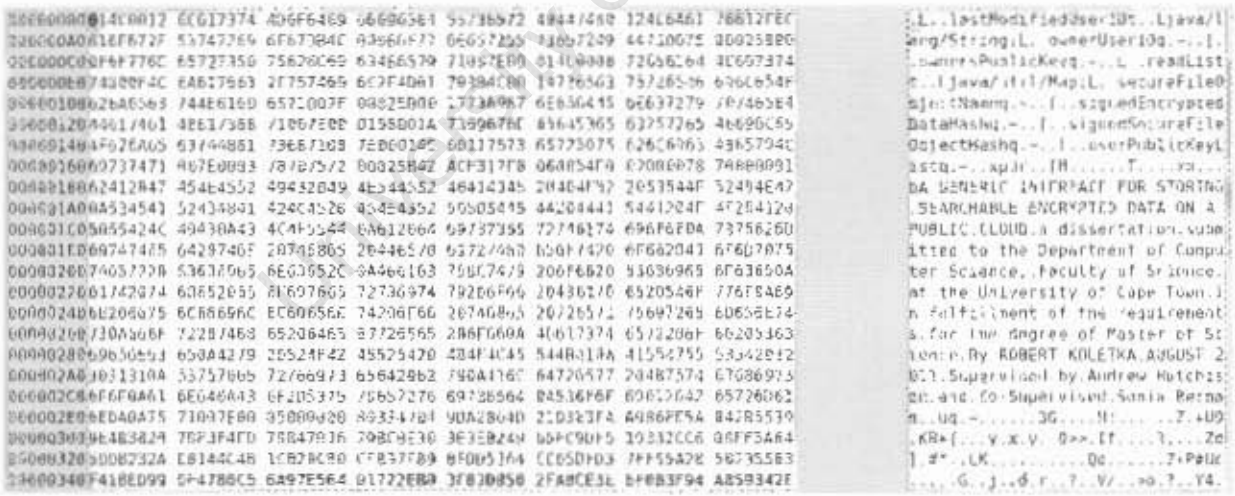


Figure 6.1: A Secure File Object without encryption serialized to the file system

The output from the hex editor is shown in Figure 6.1. The figure shows that it is easy to examine the contents of a serialized object. We then needed to test whether our scheme would be able to keep data confidential, and this was acheived with following steps:

C2

1. Create Empty Secure File Object with block encryption enabled
2. Load contents
3. Serialize the object to a file
4. Open serialized file with a text editor

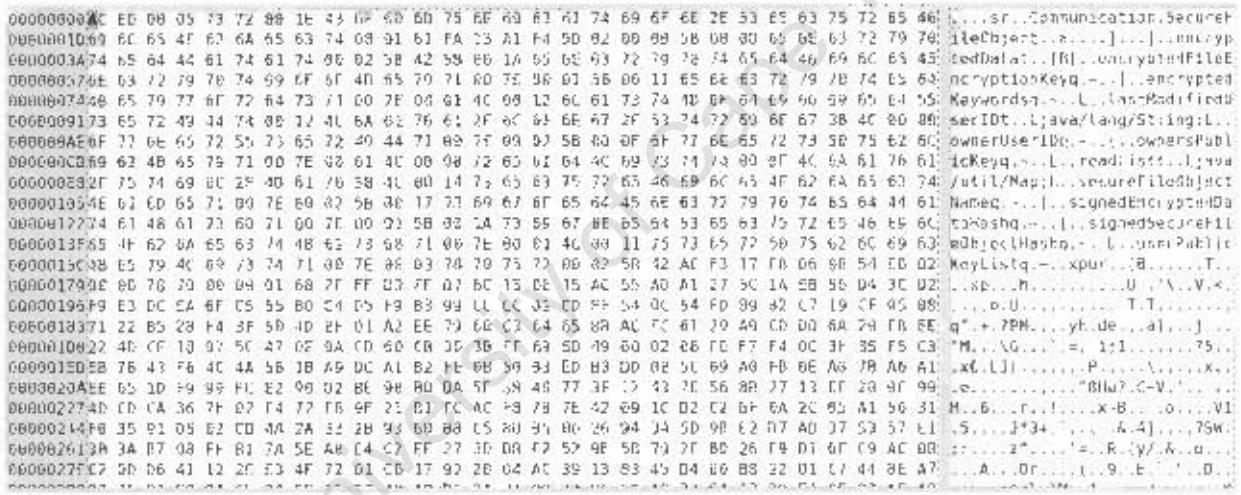


Figure 6.2: A Secure File Object with encryption serialized to the file system

The output from the hex editor is shown in Figure 6.2. This time it is not possible to read the contents of the serialized object. From these tests we can deduce that our scheme is **Successful** in keeping user data confidential.

6.2.5 Integrity

Secure File Object integrity is important in identifying whether a file has been tampered with. To test whether our scheme can detect loss of integrity we performed the following steps.

I1

1. Alice creates a new Secure File Objects, loads data and creates a signed hash.
2. Alice then makes a change but does not generate a new signed hash.
3. Alice then executes the "check integrity" command which generates a hash of the current data and compares it with the hash that is stored with the Secure File Object.
4. Since the hashes differ, that means that there has been a changes to the data.

This test has shown that our scheme can **successfully** detect losses in integrity.

6.2.6 Searchability

One of the key features of this system is allowing users to search for keywords that are attached to encrypted files stored on S3. To test this functionality, a number of files where created with overlapping keywords attached.

S1

We hard coded that File1 will have Keyword1,Keyword2,Keyword3
File2 will have Keyword2,Keyword4,Keyword5
File3 will have keyword5

Function	**Fulfilled**	**Notes**
User Functions		
File Sharing	Yes	Correctly allows users to have different access rights. Tests: *FS1, FS2, FS3*
Key-Revocation	Yes	Correctly removes a users with read rights. Test: *KR1*
Compromised Key-pair	Yes	Simulated by doing a key revoke followed by re-adding that users rights. Test: *CK1*
System Functions		
Confidentiality	Yes	Keeps data confidential when stored. Tests: *C1, C2*
Integrity	Yes	Identifies when a file has been tampered with. Test: *I1*
Searchability	Yes	Correctly returns the files containing the specified keywords. Test: *S1*

Table 6.1: Functional Specification Testing, shows the functions as defined in the Design Requirements Chapter and summarizes the results.

We then submitted a query for keyword5 and knew that File2 and File3 had to be returned as results, similarly keyword2 returned File1 and File2. We ran these tests and the results we received were the ones that were anticipated.

From these tests we established that our scheme **successfully** returned the correct results.

File Size	%Encryption Time	%Data Sign	%Object Sign	Total Time (ms)
1Kb	35.63	31.42	32.95	13.05
10Kb	37.82	30.18	32	13.75
100Kb	54.32	22.27	23.41	22
1Mb	75.19	10.16	14.65	97.95
10Mb	76.67	6.83	16.5	934.75

Table 6.2: File Encryption, Data Signing and Object signing times

6.3 Performance Testing

It is important to know whether a system works at a functional level, but equally important to determine how well the system works from a performance perspective. With a file storage system, it would not be practical for it to take up a lot of time to perform basic file system operations such as save or retrieve. For this reason it is important to check what overhead such a system adds.

6.3.1 Encryption Time

The time the system takes to encrypt is important as this indicates how long a user has to wait while the system is doing the processing. This test involved looking at how long it takes to encrypt the data, how long it takes to sign the data and how long it takes to sign the whole object to ensure the performance overhead of securing an object is tolerable. The data is displayed in Table 6.2 showing the time taken for each operation as a percentage of the total time. The input files are of sizes 1Kb, 10Kb, 100Kb, 1Mb and 10Mb. To test this we performed the following steps.

1. Create an empty Secure File Object.

2. Load contents and measure the length of time taken by this operation.

3. Generate signed data hash and time and measure the length of time taken by this

operation.

4. Generate a signed hash of the entire Secure File Object and measure the length of time taken by this operation.

This process is illustrated in Figure 6.3.

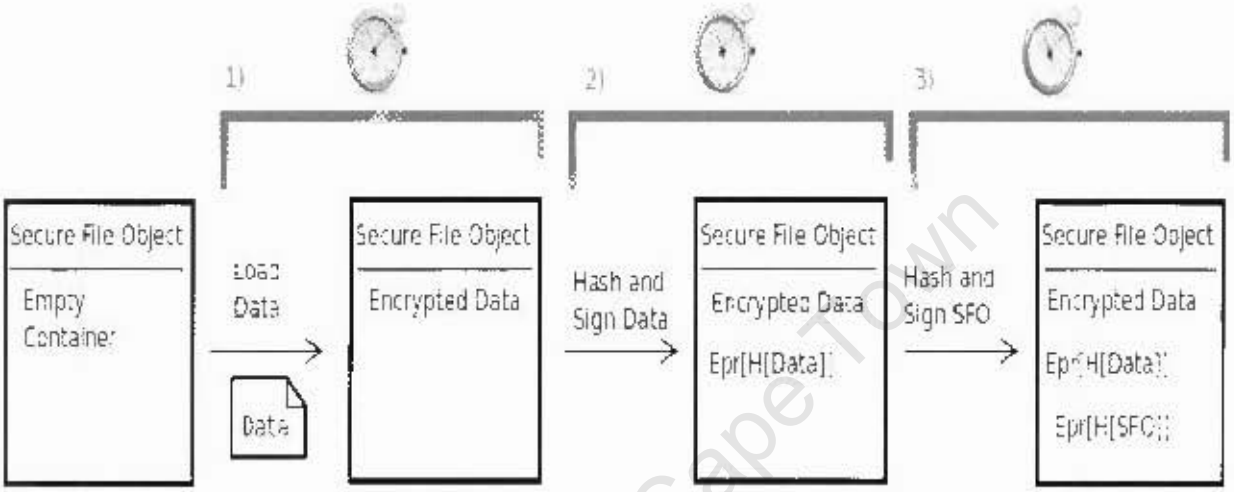


Figure 6.3: The figure shows the steps in measuring the encryption time of a Secure File Object. The first step is to measure the time taken to encrypt the data, the next step is to measure the time taken to generate a signed hash of the data and lastly the time taken to generate a signed hash of the Secure File Object.

6.3.2 File Size Overhead

In testing this scheme it is important to examine the storage overheads that such a system will incur. To determine these overheads we looked at a large number files of varying file size ranging from 1Kb to 60Mb. The files consisted of content that users would use on a daily basis, such as text files, documents and media files.

For each input file an SFO was created. The input was encrypted using DES and stored within the SFO. The sizes of the various static[1] fields within the SFO were measured. We limited the number of keywords to five using keywords of five characters in length. The filename size was also fixed at 9 characters. The total static overhead for each file was 902 bytes regardless of the input file size. The DES algorithm uses padding when doing its encryption, meaning that encrypting some input added an additional 0 to 8 bytes of overhead depending on the number of blocks. Figure 6.4 shows the security overhead as a percentage of the input file. As is shown in the graph, the overhead becomes negligible as the input size exceeds 100kb.

6.3.3 Secure Put vs Unsecure Put

Another useful test is to assume that users are already using cloud storage. For users already using cloud storage, we tested the overheads that such users would experience using our implementation. The test consisted of creating an SFO with varying input sizes and uploading these SFOs to the cloud storage service. The file sizes tested were 1Kb, 10Kb, 100kb, 1Mb and 10Mb as is shown in Figure 6.3.3. The file-set upload was repeated 30 times and the time shown in Figure 6.3.3 is the averaged time. The upload was repeated 30 times so as to smooth the data due to changes in network latencies.

Figure 6.3.3 shows the time taken to upload the files to a cloud provider from a client, it also shows the time taken to upload the files from EC2 to S3.

6.3.4 Search time

To test the search time we had considered a number of permutations of the *number of files* and the *number of keywords* per file. For each permutation, a number of 1Kb files were uploaded with a number of keywords attached. The keywords used were taken from a

[1] A Static field is a field whose doesn't change when the input is changed or does not add to the security of the SFO, such as File names.

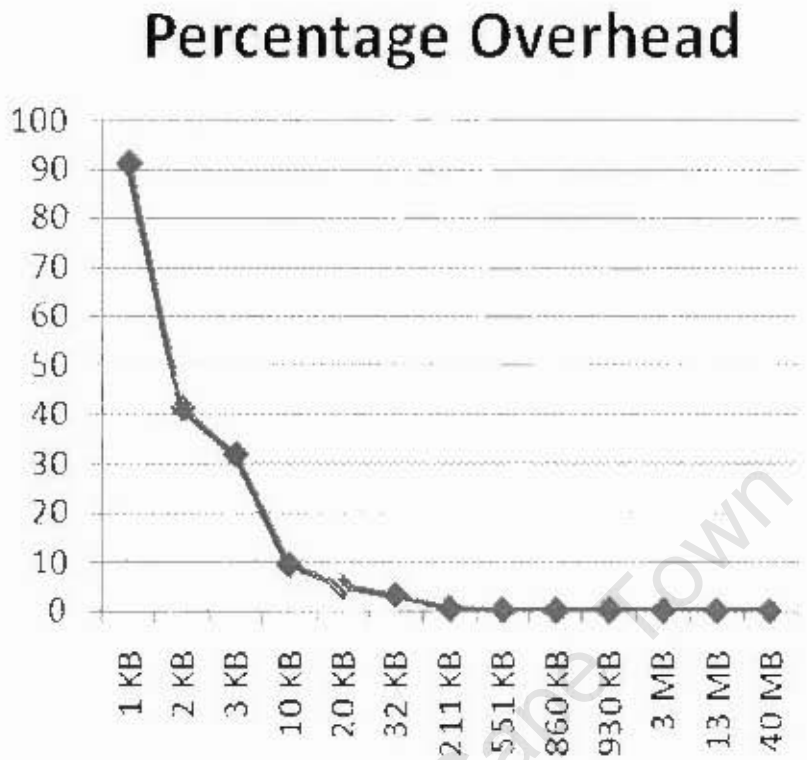
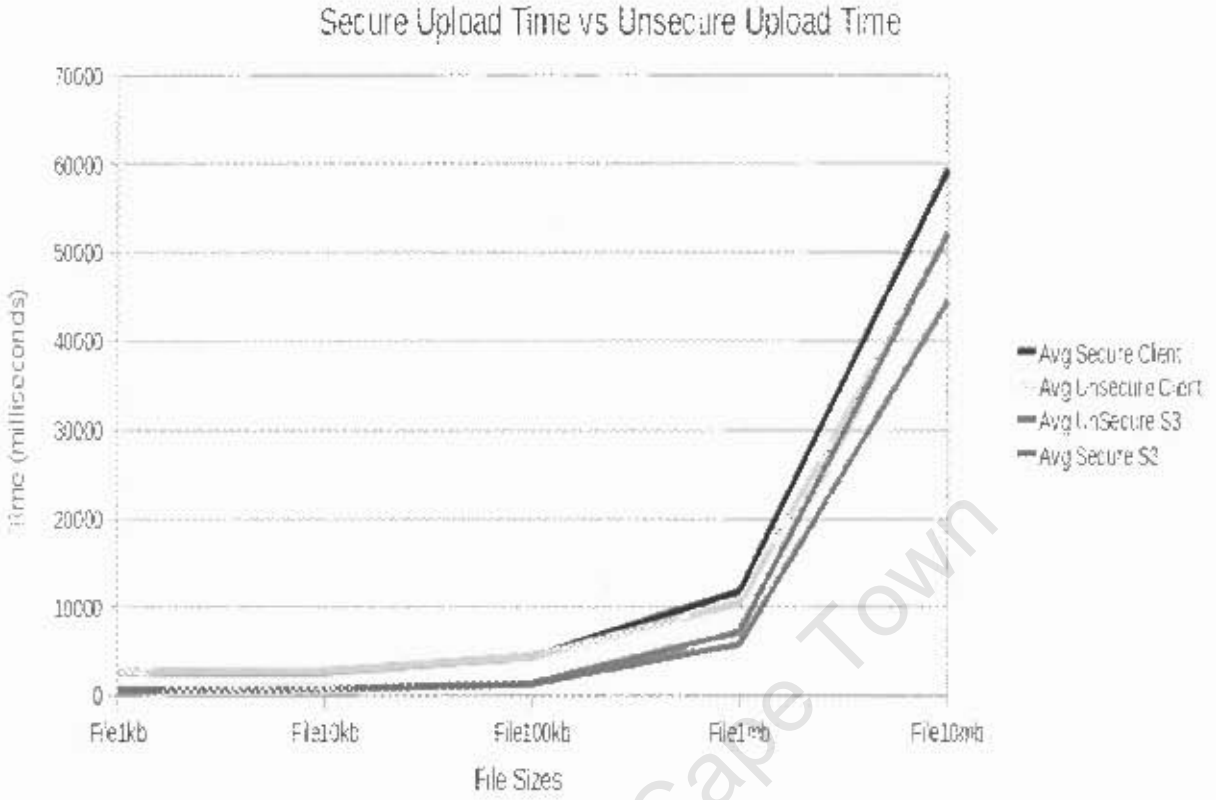


Figure 6.4: The physical storage overhead expressed as a percentage of the input file size. It is shown that as the file sizes are increased then the overhead becomes negligible.

dictionary then stored in a list to be used for the query. Once uploaded the system would then randomly select a keyword from the stored list and submit it to the EC2 service. The EC2 service then performed a search through all the files in the S3 bucket returning the matches. Figure 6.5 shows an overview, at an architectural level, of the tests performed.

Client Performance

In testing the client performance, we uploaded a number of files each with a static number of keywords attached and then tested the response time, this was repeated 30 times. Such a test gives an indication of the usability impact. Users will not be willing to wait long



periods of time for a response from the server.

The results of this test are shown in Figure 6.6. The graph shows how the response time increases as the number of files in an S3 bucket increases. The reason for the drastic increase in response time between forty files and one hundred files is due to network latencies within Amazon's infrastructure when downloading one hundred keywords file from S3 to EC2 as opposed to only downloading forty files.

Server Performance

Although testing the client performance is needed to examine how long users need to wait for responses, it doesn't accurately show the performance of the search algorithm. As the number of keywords increases the response time remains constant this is because the differences are distorted by latency changes. To accurately test the performance degradation

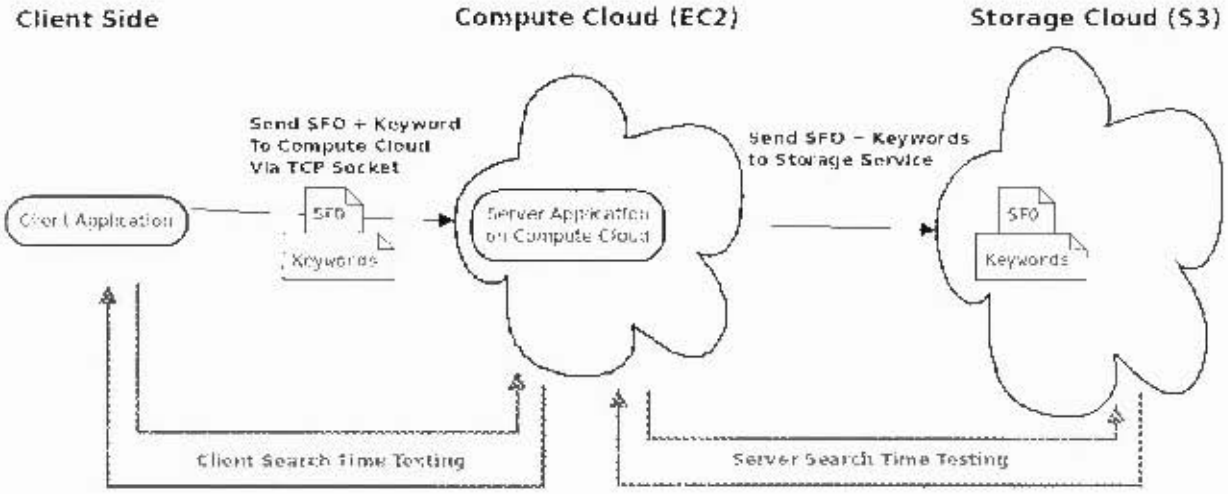


Figure 6.5: The test environment overview for the testing performed at the client and the server. There needs to be two levels of testing, from the client side and from the server side. The client side includes latency issues and examines how long a client has to wait for a search query response. The server side examines the performance of the actual search query eliminating latency issues.

as the number of keywords increases we downloaded all the files from S3 into an array in the EC2 instance's memory, thus removing latency issues and reducing the number of page faults that could occur by using other data structures. Once the files were in memory the application iterated over the files executing the search function on each item.

We initially started the testing by using a dictionary to select the random keywords that were attached to the Secure File Objects. The random keywords were then saved to a file so that we could use them later for the the search. The tests would randomly select a file from the "used keywords" file and submit this to the EC2 instance, this was repeated twenty times. The Server, running on the EC2 instance, would then measure how long it took to iterate over each keyword within each file. These first tests were timed using millisecond precision and the results of these tests are shown in Figure 6.7.

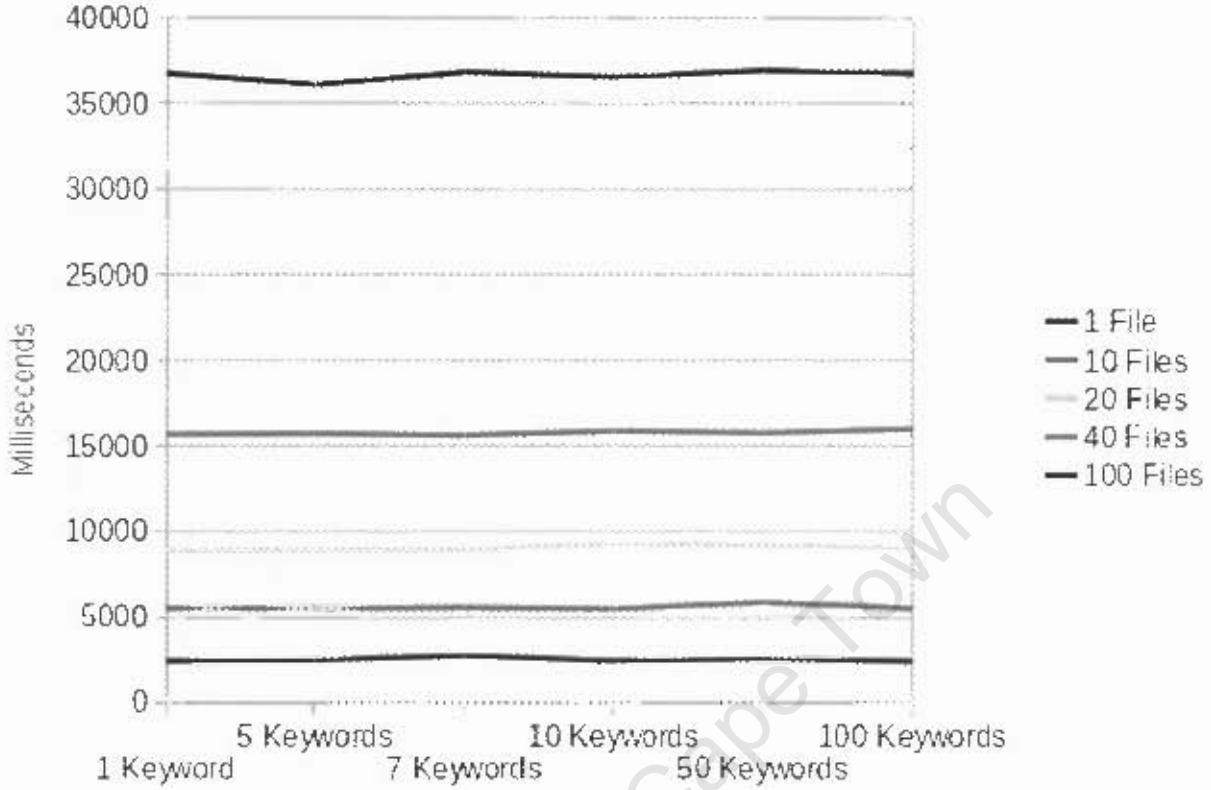


Figure 6.6: The average time taken for a client to get results back from the server application. The graph shows the time taken for varying numbers of files stored in an S3 bucket and varying numbers of keywords.

The problem with doing twenty iterations of each test case was that we were getting some unexplainable results when comparing the actual results with the theoretical analysis of the algorithm. We then decided to try nanosecond precision when doing the twenty iterations of the test cases, since some of the results we were getting were in sub-millisecond performance. However as shown in Figure 6.8 there were still unexplainable results coming from these tests. The most obvious anomaly is where the performance gets better from 100 files with 50 keywords to 100 files with 100 keywords. Obviously 5000 compare operations should take less time to complete than 10000. For this reason we decided to perform 100 iterations of each test case, hoping to get less noisy data. The results of this are shown in

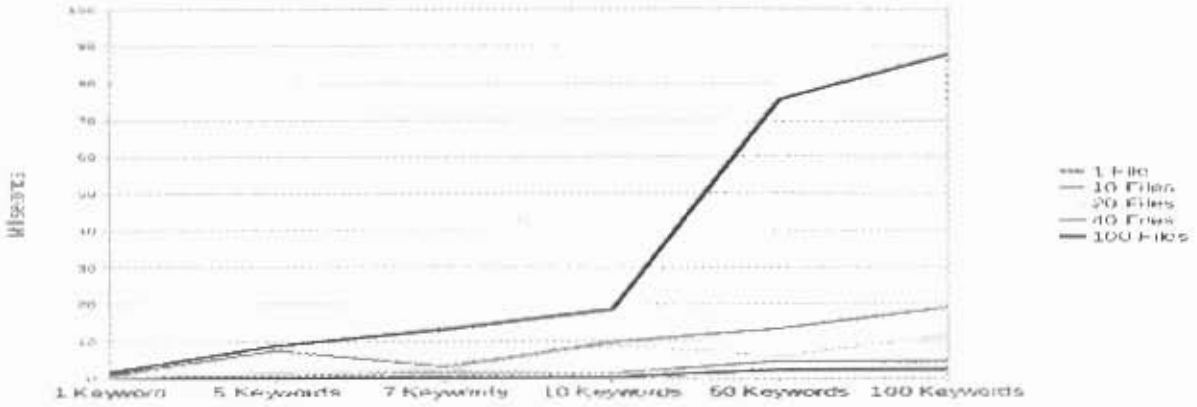


Figure 6.7: This figure shows the performance degradation of varying the number of keywords for a fixed number of files where only 20 iterations were done.

Figure 6.9 and as can be seen they are fairly different to the twenty iteration results. Another change that was made was to look for keywords that were not used when creating the test bed. This would force the algorithm to perform compare operations on all keywords in all files as opposed to searching through the keywords of a file until a match is found thus testing the worst case.

Referring back to the uses cases again, a music service could store music files in folders based on the artist. So it was fairly safe to assume that you could get folders with 10 files or even up to 100 files. We may assume that on a personal computer, in general users will not have more than 100 files in their folders.

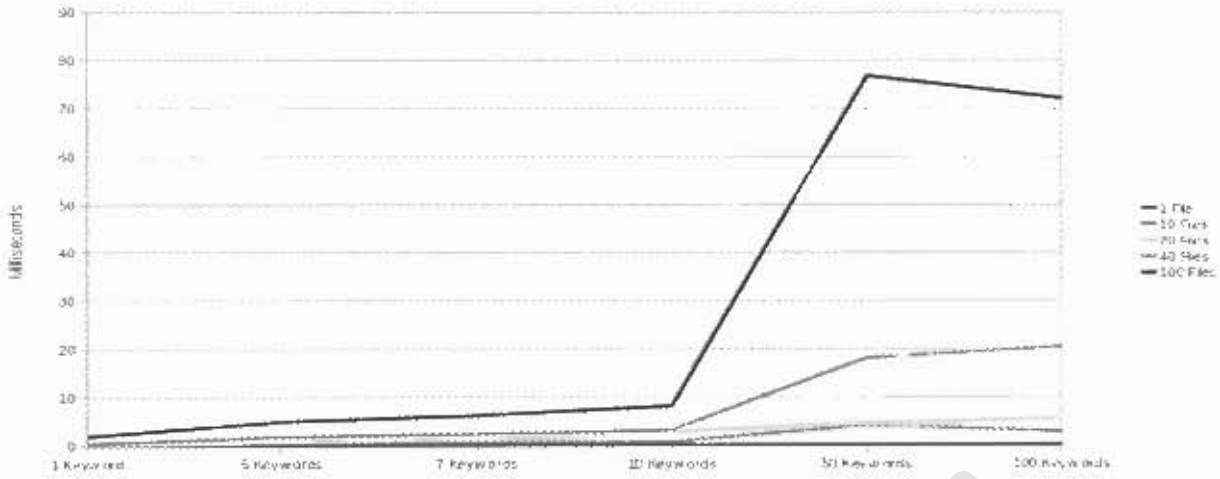


Figure 6.8: This figure shows the performance degradation when doing 20 iterations of each test case with nanosecond precision, note the anomaly happening from 100 files with 50 keywords to 100 files with 100 keywords

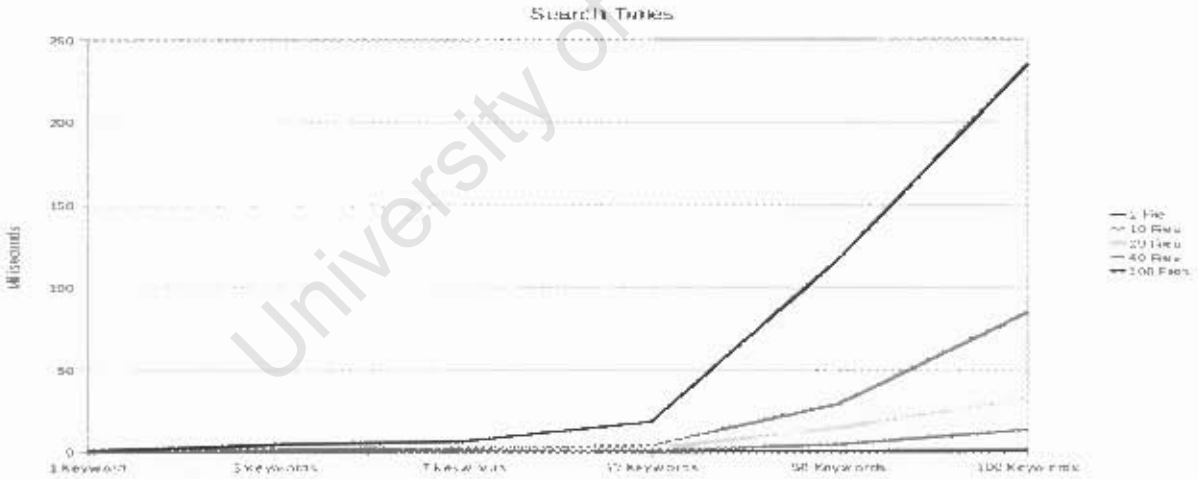


Figure 6.9: This figure shows the performance degradation of varying the number of keywords for a fixed number of files, where the test cases were repeated 100 times with nanosecond precision as well as testing the worst case each time.

6.4 Deployment Case Study

As described in Section 5.1.1, the client running on the users computer is able to accept commands via a socket interface from other applications that wish to upload data securely to the cloud. In order to test that this we decided to use the Linux email client Alpine to interface with the client interface and back-up an email securely to Amazon S3.

This was achieved by writing a Python script that accepts all the arguments given by Alpine through stdin. The script parsed the arguments for the subject of the email and then wrote the arguments to a temporary file. The script then connect to the client interface via a socket and sends a list of commands in the same format as specified in Section 5.1.1 and then passed the arguments to the "sendmail" application in /usr/sbin/ to send the email. The final step was to make Alpine execute this script, this was achieved by modifying the .pinerc configuration file, in the home directory, which is used by Alpine. We had to change the 'sendmail-path' in the configuration file to use our custom script instead of the default one. The overview of the different communicating components is illustrated in Figure 6.10

In implementing an interface for other applications we have been able to demonstrate the this system can be used as the use cases specified in Section 3. The communication between other applications and our implementation happen with the use of unencrypted sockets. There is no reason to encrypt the communication between an email client and our application since the communication is happening on the users computer which is assumed to be secure. Connecting Alpine with our application was a relatively simple task requiring a script with a few lines of code and one modification of the configuration file that is used by Alpine.

By having a simple interface it has been fairly straight forward for other applications

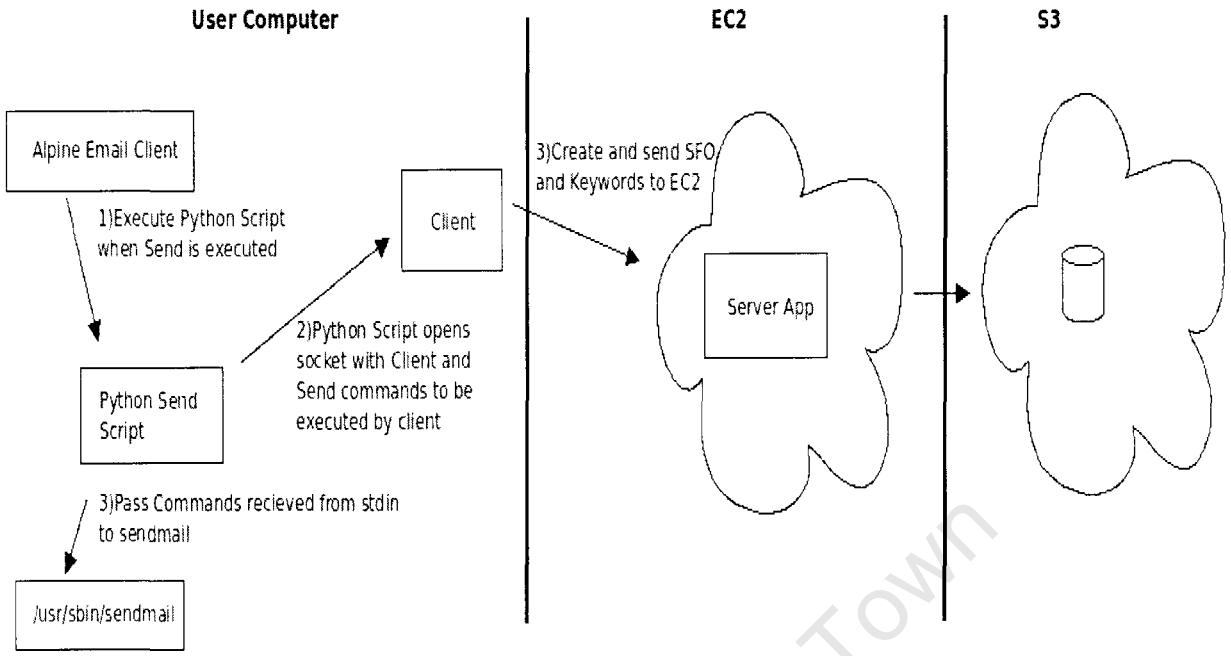


Figure 6.10: The Figure illustrates the overview of the Deployment case study and how Alpine was used to communicate with our implementation to achieve a secure backup on S3.

to interface with the client. In our testing we did not need to recompile the application that was interfacing with our client. Even if it were necessary to modify the source code, these would be minor changes since all that is required is to open a socket and send across the parameters as shown in Section 5.1.1.

6.5 Conclusion

This chapter set out to test the functional and performance dimensions of the implementation. At a functional level, high level functionality was tested such as whether the data is confidential, can the implementation detect unauthorized changes to data, does the search functionality work correctly, can file owners share data and managed user keys and lastly can the implementation recover from a compromised key-pair. At a more detailed level, the performance of the functionality was tested using different parameters. This meant observing the time it took to encrypt files of various sizes, examining the security overhead of the design, the time taken to upload files securely versus unsecurely as well as the time taken to respond to search queries using varying numbers of files and keywords. This chapter has shown that the design of such a system can be implemented and satisfies all the design requirements. To end the chapter we performed a case study illustrating how the Linux email client, Alpine, was modified to interface with our client to provide secure cloud back-up functionalitly. The case study showed that providing such functionality to applications was a trivial task, allowing for our implementation to be used in a real world setting.

Chapter 7

Analysis

In the previous chapter, we tested our implementation at both functional and performance levels. The chapter considered whether the implementation succeeded in meeting the design requirements as specified in the Section 3 and how this functionality performed using different parameters.

The purpose of this chapter is to examine the results. We assess the implication of successfully implementing all the requirements at a functional level. The performance of these functions is then analyzed starting with the *Encryption Time*, *File Size Overhead*, *Secure and Unsecure Puts* and *Search* functionality of the system. We then go on to discuss the *Current Approach* of our solution and the success of using this approach. We then analyze our results with reference to the use cases specified in Section 3 and we conclude the chapter by discussing the *Client Interface*.

7.1 Functionality

We tested the system at a functiona level to assess whether the implementation has satified the requirments specified in Section 3.1. based on the testing performed in Chapter 6, it is asserted that at our implementation successfully fulfills the requirements.

The reader should recall that were introduced in Chapter 3 as an adaptation of the list of requirements that are needed by a security storage system as specified by V.Ker et al.[20]. By fulfilling these requirements it is implied that our system is secure by their

standards.

7.2 Performance

It is important to test whether a design can be successfully implemented but equally important is to test how well the implementation performed. For this reason the system was exercised under various load functional conditions and tested how long various operations took to complete. Tests considered how long it took for data to be encrypted and signed, the size of the security overhead that was added, the time it took to upload a file to the cloud with and without security and lastly the time it took for the implementation to process search queries.

7.2.1 Encryption Time

In order to examine the performance of the encryption time we looked at three operations that would ensure that the data has been secured.

1. Loading the data from a file and encrypting it using the DES encryption algorithm.
2. Hashing the encrypted data and signing it with the file owner's private key
3. Generating a hash of the entire Secure File Object and signing that with the owners private key

We measured the time taken for each of these operations and expressed them as a percentage of the total time taken to perform all three operations. The data is displayed in Table 6.2 and is shown in Figure 7.1. The table shows that the time taken to perform all three operations took 13.05 ms on file of 1Kb in size, with roughly a third of the time being used for each operation, similarly for a file with 10Kb. This changes for file sizes greater than 10Kb where the dominant operation is encryption. This is too be expected as the time taken in generating a hash and signing it will not increase as much as doing a

symmetric encryption on the data when the size of data is increased. When examining the raw data, the time taken to encrypt a 100Kb is 11.95 ms and 73.65 for a 1Mb file which is a 516% increase. The time taken to perform a hash and sign on a 100Kb is 4.9 ms and 9.95 ms for a 1Mb, which is only a 103% increase, and a 178% increase when looking at the hashing and signing of the Secure File Object. Figure 7.1 shows how the encryption time becomes more dominant as the file sizes are increased.

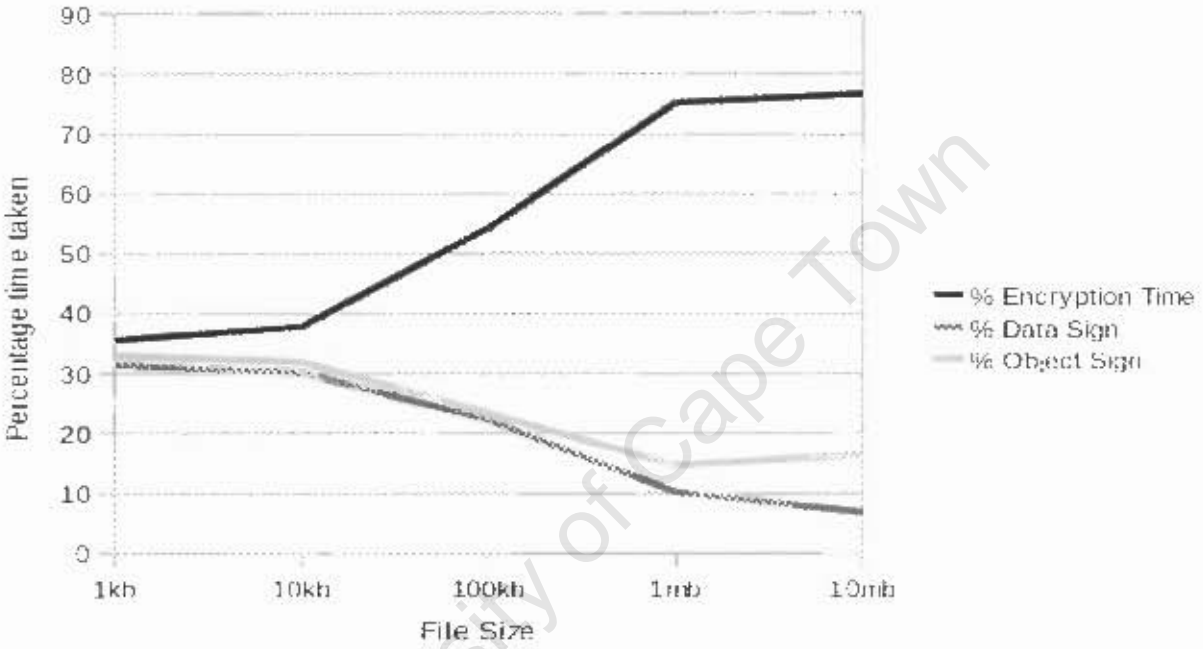


Figure 7.1: The Figure illustrates the percentage of time spent doing each operation in encrypting the data, signing the data hash and signing the Secure File Object hash.

7.2.2 File Size Overhead

These tests aimed at finding what the physical overhead of the design is. Figure 6.4 shows the overhead for a large number of files of varying sizes. It is shown that regardless of the input size, there is always a static overhead of 902 bytes and a maximum of 8 bytes which

is used by DES for padding. The total security overhead never exceeds 910 bytes. This makes the system efficient when using file sizes of more the 10Kb by adding less than 1% in overheads. This is assuming that there is only one read / write user. To add a read users means adding an entry to the read list which adds an additional 136 bytes per entry and 171 bytes for an entry in the Public Key list.

Such overhead can seem reasonable when looking at the use cases specified in Section 3. An emailing system that backs-up files to S3, a 902 Byte overhead would not be significant given that such a file would have a number of emails contained with in, which can easily grow larger than 10Kb. The sample saved message that is used by Alpine[1] is roughly 500 bytes for the header and 3 lines of text. For this system to provide an under 10% overhead we would need 2 of these sample emails. In the case of a Dropbox type of an application, users would upload general documents, to media files. All of which can be range from 1Mb through to 1Gb in size resulting in an insignificant overhead.
The music system use-case would also result in a less than 1% security overhead, this is because a typical mp3[2] song of length 4 minutes is roughly 5Mb in size. An overhead of 902 bytes for a file that is 5Mb big results in an overhead of less than 0.018%.

7.2.3 Secure Put vs Unsecure Put

The reason for testing Secure Put versus Unsecure Puts is to examine the overhead this system would place on users already using the cloud, specifically S3, to store their documents. It did not make sense to look compare the performances of local storage with that of cloud storage since these two will obviously differ greatly due to network transfer speeds. Figure 6.3.3 shows the time it takes to upload from the client to EC2, these are the *Avg Secure* and *Avg Unsecure* values. The next step was to examine how long it takes to upload from EC2 to S3, these are the *Avg Secure S3* and *Avg Unsecure S3*.

[1] A Linux email client, `http://www.washington.edu/alpine/`

[2] `http://en.wikipedia.org/wiki/MP3`

As is shown in the graph, the Avg Secure and Avg Unsecure values follow closely together, this is because the time taken to upload a 10Kb and a 10Kb + 902b file are very similar. The only differences that affect the values are the changes in network load and latencies which makes it difficult to determine the exact meaning of the slight differences because there are network changes happening in the local network as well as within the Amazon Web Services network. It is important to note that the difference in uploading a file with this security scheme and without scheme is small. In the case of the 10Mb file, it was a 14% increase in upload time, whilst in the the case of the 1Kb file it was only a 0.5% increase.

7.2.4 Search

In testing the performance of the search functionality, we broke it down into two broad categories. The search as seen from the clients perspective, with network latencies, and the performance at the server, done in such a way so as to eliminate network latencies and to simply capture the time the actual search took. This was achieved by loading all keyword files into memory from a given bucket, then timing how long it takes to iterate over all the items to find the matches.

Client

The client data is displayed in Figure 6.6. As is shown, from the clients perspective, the number of keywords attached to the files doesn't degrade the performance of the search. The only significant contributor to performance degradation is the number of files in a given bucket. It can be seen from the graph that as the number of files increases so does the the response time, nearly proportionally. The increase from 40 files to 100 file is an increase of 150% in the number of files and the response time increases by 133%. Similarly the increase from 20 files to 40 files is a 100% increase in the number of files and the response time increase is 74%. The reason for the this performance degradation is because

the server running on the EC2 instance needs to download the keyword files off S3, if there are 100 files it needs to download 100 keyword files. The latency from the client to the server remains fairly constant so it is this downloading from S3 which slows down response times.

Let us apply these results to our emailing use case as explained in Section 3. Should this emailing system have 40 files in an S3 bucket and it decides to submit a query, it will take roughly 15 seconds for it to receive a response from the server regardless if there is one keyword per email or 100.

Server

Although it is useful to look at client response times, as mentioned these results include network latencies which do tamper with the search response times. For this reason we decided to test the efficiency of the search algorithm without any external influences.

The results of the testing are shown in Figure 6.9, Figures 6.7 and Figures 6.8. Figure 6.7 displays the results when fixing the number of files but increasing the number of keywords using millisecond precision and 20 iterations. Figures 6.8 shows the results of using nanosecond precision with 20 iterations whilst shows the results of using nanosecond precision with 100 iterations Figure 6.9.

Algorithm Complexity The algorithm has a worst case complexity of **O**(KN), where K is the number of keywords per file and N is the number of files. A simple check shows us that using 20 iterations with millisecond precision Table 7.1 gave incorrect results. If we look at the performance of 100 files with 10 keywords this results in **O**(1000) search operations, which resulted in about 19 milliseconds of computation. If we look at the results of 40 files and 100 keywords this results in **O**(4000) search operations with a similar time. This seemed suspicious because 4000 search operations had a similar time as 1000

Parameters	Operations	Theoretical (ms)	Actual (ms)
20 Iterations, Millisecond precision. Baseline 0.25.			
10 Files 100 Keywords	1000	250	4.4
100 Files 10 Keywords	1000	250	18.55
40 Files 100 Keywords	4000	1000	19.05
100 Files 50 Keywords	5000	1250	75.6
100 Files 100 Keywords	10000	2500	87.75

Table 7.1: This table shows the performance of executing 20 iterations with millisecond precision.

search operations. We then tried to perform the testing using nanosecond precision with 20 iterations but this also gave strange results most obviously the drop in search time with 100 files and 50 keywords to 100 keywords. It again seemed strange that 5000 search operations took longer than 10000 search operations. The reason for this could not have been due to page faults in memory since all keyword files were stored in an array, the probability of page faults was small. The reason for this noisy data is that there are a lot of underlying operations happening behind the scenes such as garbage collection, context switches and other Java / operating systems tasks. For this reason we decided to do more iterations of each test case to get less noisy data. These results are shown in Figure 6.9, where the results are more intuitevely what we would expect.

When examining Figure 6.9, we notice that these results are more appropriate and are closer to the theoretical analysis. With these new results, we notice that 100 files with 10 keywords each takes approximately 20 milliseconds where as 40 files with 100 keywords is approximately 80 milliseconds. This seemed intuitively correct since 4 times the number of compare operations should take roughly 4 times the amount of time. Our predictions are also more accurate when using 100 iterations.

Parameters	Operations	Theoretical (ms)	Actual (ms)
20 Iterations, Nanosecond precision. Baseline 0.066.			
10 Files 100 Keywords	1000	66	2.94
100 Files 10 Keywords	1000	66	8.16
40 Files 100 Keywords	4000	264	20.54
100 Files 50 Keywords	5000	330	76.73
100 Files 100 Keywords	10000	660	72.06

Table 7.2: This table shows the performance of executing 20 iterations with nanosecond precision.

Testing with 20 iterations and millisecond precision When performing 20 iterations with millisecond accuracy our baseline was 0.25 milliseconds, the baseline is where there is only one file with one keyword. The baseline when using nanosecond precision and 20 iterations was 0.066 milliseconds and 0.032 for the 100 iterations results. A comparison of the actual results versus the theoretical results are shown in Table 7.1. The theoretical results are simply the baseline multiplied by the number of comparison operations. As can be seen, there is a large difference between the theoretical and actual results in the 20 iteration millisecond precision results. There is also a large difference between the search queries for 10 Files 100 Keywords and 100 Files 10 Keywords, both of which require 1000 search operations. Similarly, the increase from 4000 operations to 5000 operations is only 25% while the actual performance deteriorates by nearly 400%.

Testing with 20 iterations and nanosecond precision When examining results of using nanosecond precision with 20 iterations, the results improve slightly in certain situations but are still far off the theoretical analysis as shown in Table 7.2. There is also an anomaly where 100 files with 100 keywords each performs better then 100 files with 50 keywords each, this seems incorrect because the former is performing 10000 compare

operations whereas the latter is only performing 5000 compare operations.

Testing with 100 iterations and nanosecond precision The last set of results are by far the best, the theoretical performances match the actual results gathered far better than in the other two tests as is shown in Table 7.3. As mentioned the number of operations logically affects the time taken to complete the search. The time taken to complete a search over 10 files with 100 keywords and a search over 100 files with 10 keywords is nearly the same, only a 37% difference, where this difference was over 300% and 200% for the other to tests. If we try to use the theoretical analysis along with the baseline to try and predict actual results, these predictions are much better than in the two other cases. If we try predict how long 10000 operations, we get a theoretical time of 320 milliseconds and the actual result is 335 milliseconds, only a 4% under estimate. Whereas the difference in the other two tests was a 2774% over estimate and a 816% over estimate.

The conlusion of this discussion is that it is neccessary to perform at least 100 iterations with nanosecond precision to have the actual results closely match the theoretical analysis.

7.3 Use Cases

An email client with the benefit of backed up storage in the cloud is considered a good test case for use with an SFO, in that around 10 keywords is seen as sufficient for folder / description. Similarly it is considered that this could also be appropriate for a file system or music library augmented with tags. These use-cases are well with in the design parameters. If each bucket has 100 SFO's with emails as contents and 10 keywords each, then a query to a bucket by the emailing application would only take about 18.3 milliseconds. However

Parameters	Operations	Theoretical (ms)	Actual (ms)
100 Iterations, Nanosecond precision. Baseline 0.032.			
10 Files 100 Keywords	1000	32	13.3
100 Files 10 Keywords	1000	32	18.3
40 Files 100 Keywords	4000	128	84.95
100 Files 50 Keywords	5000	160	117.49
100 Files 100 Keywords	10000	320	335.13

Table 7.3: This table shows the performance of executing 100 iterations with nanosecond precision.

should the emailing application only need 5 keywords per file then queries will only take 4.6 milliseconds, ultimately there will be a trade-off for each use case.

7.4 Current Approach

As mentioned in Section 4.5, the technique used to add secure searching is an adaptation of the method used developed by Waters et al.[30]. In their symmetric approach, they stored the encryption key with each keyword entry in their audit log. This would mean that we would have to store the *File Encryption Key* with each keyword entry, which would be inefficient and incorrect since granting a user search access does not imply that the user should have access to the data. In our approach we replaced the key with another random bit string. In their paper they found that using their symmetric encryption scheme was not secure since if the audit log server was compromised then an adversary would have access to the secret key used in the secure keyword creation. This was not the case in our setting since the these secret keys are stored on the client, and are encrypted using the owners secret key.

The structure of the Secure File Object was adapted from the data structure as used by Boneh et al [12]. The difference in our scheme is that we have two lists to maintain

read / write access, key-revocations and perform integrity checks. The first list is the read access list and then there is a public key list that is used to perform integrity checks by retrieving the users public key and using it to decrypt the signed data hash. The method used by [12] is to have an entry for each user, if a user has write access then the private part of the File Signing key is stored. Storing multiple copies of the private part of any asymmetric key can be costly with regards to the space required.

7.5 Approach success

The approach was successful overall in that it fulfilled all the requirements that were identified in Section 3. This means that users are able to securely store data on a public cloud and detect whether there have been any unauthorized changes to the data. File Owners can also allow other users different levels of access to the data stored as well as revoke rights to certain users as is done on a traditional computer. The implementation also allows for users to search through encrypted data in such a way that the cloud provider is not able to determine the contents of the data stored. Thus from a functional level, the design removes the need for trust from a cloud provider. Where this approach lacks is in preventing malicious users from deleting files or modifying files as discussed in Section 4.2.1. Should an adversary gain access to the account, then files can be deleted. This implementation can detect such action by storing a local file list and doing a reconciliation, however it cannot restore the deleted file or prevent file deletion.

The searching functionality of the design also proved to be successful from both a functional level and a performance level. The implementation successfully found all files containing the keywords in a search. Our testing also managed to match up the theoretical model with the actual results to within 4% difference. Where the searching would start to fail is with very large data sets. Using our theoretical model we estimate that 1'000'000

Files with 100 keywords would take approximately 8 hours to compute. However a more efficient indexing scheme would give far better performances for large datasets.

7.6 Conclusion

This chapter set out analyze the implementation at both a functional and performance level. Our implementation has successfully fulfilled all the requirements that we set out to achieve as specified in Section 3. We have shown that the overheads incurred by these requirements are reasonable. As the size of a file increases, the dominant operation in securing the file becomes the encryption and we have shown this by encrypting a number of files with increasing size and graphically showing that as the file size increases so does the dominance of encrypting the data. Our implementation has a very small data overhead, only adding 910 bytes of data where there is only one user of the file. This results in less than 1% in data overheads for files greater than 10Kb. We showed that, since there is a minimal increase in the data size when using our implementation, this meant that the time taken to upload a secure file and a non-secure file would be very similar. The search functionality of our implementation was then analyzed, we explained the reasons for performance degradation at both a client and server level, we also performed a theoretical analysis of our search algorithm and showed that our actual results converged closer to the theoretical prediction as we performed more iterations of the test cases. We then discussed the possible use cases of the system and how other applications can interface with it. And lastly we discussed how our approach has been adapted and differs from other techniques used in securing untrusted distributed storage systems, and then discussed the success of our approach and a few possible drawbacks.

Chapter 8

Conclusions and Future work

The objective of the project was to design a system that would allow users to store data securely on a public cloud provider such as Amazon's Web Service, specifically Amazon's Simple Storage Service. We also wanted to add the ability to search through encrypted data to return only the files that were relevant to the users needs. The reason for doing this is that there is a lack of trust in the security of cloud providers as stated by Armbrust et al in [1].

8.1 Summary of Approach

We started out by examining distributed storage and what techniques are used to secure distributed storage where the storage is not trusted since these techniques could be relevant to our project. We then studied the various research and techniques being used to allow for encryption with keyword search and found that *Symmetric Encryption with Keyword search* was most relevant to our needs.

The next step was to list the requirements needed to make a storage system secure, and we found a survey that gave a concise list of requirements that we adapted to our setting. The requirements that needed to be fulfilled were for data *Confidentiality*, data *Integrity*, the ability for *File Sharing*, *Key Revocation* for removing read user access, the ability to recover from a *Compromised Key-pair* and lastly the need to *search* through encrypted data. There was also a need for an authentication and access control protocol

which ensured that only legitimate users were able to access the system.

We were able to fulfill these requirements with the use of a modified data structure that was adopted from Goh et al.[12] for securing distributed storage systems, named *Sirius*, that we called the *Secure File Object*. This modified data structure fulfilled all the requirements except for Searchability. To fulfill the searchability requirement of the project we adapted the *Symmetric Encryption with Keyword Search* techniques developed by Waters et al. in [30]. The data structure used for searchability, in our design was called *SFO Keywords*. At a high level each Secure File Object has an SFO keywords file object attached to it, which is used to store the keywords attached to the data. Should an encrypted keyword be submitted, the server only does the cryptographic computations on the SFO keywords files, returning the results where there is a match.

We then implemented a prototype in Java to run on a Linux system using a 1024bit RSA key pair per user. There is a client that handles all cryptographic operations on the users computer and there is a server which runs on a compute instance. This server accepts requests from the client and does the necessary computations for the search algorithm. Our data encryption was done using the DES encryption algorithm and we generated hashes of the encrypted data using MD5 for both the data fields and the Secure File Object as a whole.

This prototype was then used to perform testing which we then used to evaluate the success of the project. We found that the requirements were successfully fulfilled and the performance impact of fulfilling these requirements was acceptable. We tested the performance impact these requirements had in terms of *Data Encryption Time, File Size Overhead, Secure vs Unsecure Puts, the Client Interface and the Search Algorithm.* The Search Algorithm was tested at both the client and the server. The client search testing was done merely to examine the effect this scheme would have for users. The server search

testing was done in order to remove network latencies and transfer speeds to compare the theoretical analysis and the actual results that we were getting. We found that our scheme has a very small space overhead of about a constant 910 bytes for our test cases. Since this overhead is so small the difference in uploading unsecure files versus using our scheme is also insignificant. The data encryption time increased linearly with the size of input data, which is expected, and the time taken to respond to search queries also increased with the number of search operations needed, also as expected. We found that our measurements needed to be performed with nanosecond precision. Initially we had twenty search iterations which provided us with limited accuracy in the results but as we increased these iterations to one hundred and beyond, the actual results started to converge with our theoretical analysis, to within 4% in some cases. The deviation in the low iteration range is considered to be due to the Java and Operating System internals which are disproportionately skewed in the case of few iterations. The results of the study are of significant since they highlight the fact the secure cloud storage with search functionality can be achieved. The searchable encryption algorthms can be applied in a cloud context and the performance testing and analysis can prove benefial to future research.

8.2 Future Work

There is further work to be done with this project. A study could be performed in the searching aspect of the system, more specifically querying functionality. At the moment the design performs a brute force scan across all the keywords within all the files. One could look at various indexing techniques that could be used in an encrypted setting to perform more efficient look ups. This could be extended to allow for range scans across the encrypted keywords using indexing techniques

Another study could be performed to compare the performance of RSA and Eclypitic Curve Cryptography to determine the speed and storage overheads of each in a cloud set-

ting.

Another draw back of the system is that it is currently designed to encrypt/decrypt whole files. There is no ability to perform random access on a file. If a user wishes to modify a certain block of the file, then the entire file must be downloaded, decrypted, modified, encrypted and sent back into the cloud. Techniques could be used from secure distributed storage systems to overcome this issue.

8.3 Meeting the Objectives

We stated that the hypothesis of the study was to design a solution that would allow users to securely store data on an untrusted public cloud provider, whilst allowing for encrypted keyword search. We set out evaluate this by ensuring that the secure storage requirements were met and performed the task of examining the overheads of securing data on a cloud provider as well as the performance overheads of adding encrypted searchability. We performed the testing to evaluate the implementation of the requirements, and the impacts of them. Based on these tasks, we have found that our design proves to be efficient in both the storage overheads and the processing time added when searching through a small number of files and that all the requirements could be met.
The study has shown that secure searchable storage can be securely added in a cloud storage service.

References

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[2] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt 2004*, pages 506–522. Springer, 2004.

[3] D. Boneh and M. Franklin. Identity based encryption from the weil pairing. *SIAM Journal on Computing*, 32:586, 2003.

[4] D. Borthakur, J. Gray, J.S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime at facebook. In *ACM SIGMOD Conf*, pages 1071–1080, 2011.

[5] E.A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[6] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[7] Y.C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455. Springer, 2005.

[8] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, page 88. ACM, 2006.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):220, 2007.

[10] J. Feng, Y. Chen, and P. Liu. Bridging the missing link of cloud data storage security in aws. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–2. IEEE, 2010.

[11] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.

[12] E.J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145. Citeseer, 2003.

[13] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM, 2002.

[14] J.H. Howard et al. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 23–26. Citeseer, 1988.

[15] M. Jakl. Representational State Transfer.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, page 42. USENIX Association, 2003.

[17] S. Kamara and K. Lauter. Cryptographic cloud storage. *Financial Cryptography and Data Security*, pages 136–149, 2010.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[19] L.M. Kaufman. Data security in the world of cloud computing. *Security & Privacy, IEEE*, 7(4):61–64, 2009.

[20] V. Kher and Y. Kim. Securing distributed storage: Challenges, techniques, and systems. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 9–25. ACM, 2005.

[21] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[22] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the Cloud? An architectural map of the Cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31. IEEE Computer Society, 2009.

[23] E.L. Miller, D.D.E. Long, W.E. Freeman, and B. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, 2002.

[24] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001. Proceedings*, page 329. Springer, 2001.

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

[27] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 44. IEEE Computer Society, 2000.

[28] P. Stanton. Securing data in storage: A review of current research. *Arxiv preprint cs/0409034*, 2004.

[29] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, page 160. ACM, 2001.

[30] B.R. Waters, D. Balfanz, G. Durfee, and D.K. Smetters. Building an encrypted and searchable audit log. In *ISOC Network and Distributed System Security Symposium (NDSS 2004)*. Citeseer, 2004.

Appendix A

Appendix

A.1 Secure File Object

```
 public class SecureFileObject implements Serializable
{
    private String secureFileObjectName;
    private String ownerUserID;
    private String lastModifiedUserID;

    private Map<String,byte[]> readList;
    private Map<String,UserPublicKeyListEntry> userPublicKeyList;

    private byte[] encryptedFileEncryptionKey;
    private byte[] ownersPublicKey;
    private byte[] signedSecureFileObjectHash;
    private byte[] encryptedKeywords;

    private byte[] encryptedData;
    private byte[] signedEncryptedDataHash;

    public SecureFileObject(String secureFileObjectName, String ownerUserID,
            String lastModifiedUserID, Map<String, byte[]> readList,
            Map<String, UserPublicKeyListEntry> userPublicKeyList,
```

```
            byte[] encryptedFileEncryptionKey, byte[] ownersPublicKey)

 }
```

A.2 Secure File Object Processor

```
//The class allows the user to manipulate a given SecureFileObject instance
public class SecureFileObjectProcessor implements Serializable
{
    public SecureFileObject CreateSecureFileObject(String secureFileObjectName,
String ownerUserID,
String lastModifiedUserID, Key fileEncryptionKey,
Key ownersPublicKey, Key ownersPrivateKey)

    public SecureFileObject CreateSecureFileObject(String secureFileObjectName,
  String ownerUserID, RSAPublicKey ownersPublicKey,
  RSAPrivateKey ownersPrivateKey )

    public String GetSecureFileObjectName(SecureFileObject sfo)

    public void AddUserPublicKey(String UserID, Key UserPublicKey,
  boolean isWriter, SecureFileObject sfo)

    public void AddReadUser(String UserID, Key UserPublicKey,
Key OwnerPrivateKey,SecureFileObject sfo)

    public boolean  AddReadWriteUser(String UserID, Key UserPublicKey,
Key OwnerPrivateKey,SecureFileObject sfo)
```

```
    public void RemoveReadUser(String UserID,Key OwnerPublicKey,
Key OwnerPrivateKey,SecureFileObject sfo)

    public void RemoveReadWriteUser(String UserID,Key OwnerPublicKey,
Key OwnerPrivateKey,SecureFileObject sfo)

    public byte[] HashSecureFileObject(SecureFileObject sfo)

    public byte[] HashEncryptedData(SecureFileObject sfo)

    public void SignAndHashSecureFileObject(Key OwnerPrivateKey,
SecureFileObject sfo)

    public boolean SignAndHashEncryptedData(String UserID,
Key PrivateKey,SecureFileObject sfo)

    public boolean VerifyEncryptedData(SecureFileObject sfo)

    public boolean VerifySecureFileObject(SecureFileObject sfo)

    public byte[] SignHash(byte[] hash, Key PrivateKey)

    public void SetData(byte[] data, String UserID,Key UserPrivateKey,
SecureFileObject sfo)

    public long SetDataFromFile(File f,String UserID,
Key UserPrivateKey,SecureFileObject sfo,
```

```
boolean Secure)

    public byte[] GetData(String UserID, Key UserPrivateKey,SecureFileObject sfo)

    public void SaveDataToFile(File f,String UserID, Key UserPrivateKey,
            SecureFileObject sfo)

    public void ListUsers(SecureFileObject sfo)
```

A.3 AmazonS3

```
public class AmazonS3
{
    private String myAccessKey;
    private String mySecretKey;
    private S3Service myService;

    public AmazonS3(String myAccessKey, String mySecretKey) throws Exception

    public void CreateS3Bucket(S3Bucket bucket) throws Exception

    public void SendSecureFileObject(SecureFileObject sfo)

    public SecureFileObject GetSecureFileObject(String SecureFileName)

    public  S3Object SendObject(S3Bucket myBucket,String fileName)
```

```
    public  S3Object SendObject(S3Bucket myBucket,File tempFile)

    public S3Object SendObject(S3Bucket myBucket,String objectKey, String data)

    public  S3Object GetObject(S3Bucket myBucket, String objectKey)

    public void ListObjects(S3Bucket myBucket)

    public void DeleteObject(S3Bucket bucket, String obj_Key)

}
```

A.4 Secure File Object Keywords

```
public class SFOSecureKeywords implements Serializable
{
    byte[] randomBitString;
    byte[] flag;
    List<byte[]> encryptedKeyWords;

    public SFOSecureKeywords(byte[] randomBitString, byte[] flag,
            List<byte[]> encryptedKeyWords)

}
```

A.5 Secure File Object Keywords Processor

```
public class SFOSecureKeywordsProcessor implements Serializable
{
    public SFOSecureKeywords CreateSFOSecureKeywords(String Keywords,SecretKey kS)

    public boolean SearchForKeyword(byte[] capability, SFOSecureKeywords
    encryptedKeywords)
}
```

A.6 Symmetric Searchable Encryption

```
//The class is used to generate aa SFOSecureKeywords instance
 public class SymmetricSearchableEncryption
{

    public SymmetricSearchableEncryption()

    public SecretKey GenerateHMACShaSecretKey()

    public SecretKey GenerateHMACShaSecretKey(byte[] ksBytes)

    //The function generates a search capability for a given keyword which is then
    //used to query the encrypted files.
    public byte[] GenerateCapability(SecretKey ks, String keyword)

    //The function generates a list of encrypted keywords which is then added to a
    //SFOSecureKeywords instance
```

```
    public List<byte[]> GenerateEncryptedKeywordList(StringTokenizer keywords,
            BigInteger r,BigInteger flag,
            SecretKey kS)

    //The function search the list of encrypted keywords looking for matches
    public boolean SearchEncryptedKeywordList(List<byte[]> KeywordList,
            byte[] searchCapability, BigInteger flag, BigInteger r)

    //The function performs the cryptographic transformations to determine whether
    // a given capability matches an entry in the list
    public boolean SearchableKeyWordCompare(byte[] searchCapability,
    byte[] encryptedKeyWord,BigInteger flag, BigInteger r)

     public byte[] SearchableKeyWordEncrypt(String keyword,BigInteger r,
             BigInteger flag,SecretKey S)

}
```