An Architecture for Secure Searchable Cloud Storage

Robert Koletka
Department of Computer Science
University of Cape Town,
South Africa
Email: robert.koletka@uct.ac.za

Andrew Hutchison
Department of Computer Science
University of Cape Town
Cape Town, South Africa
Email: hutch@cs.uct.ac.za

***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.**

This paper describes a solution that allows users to securely store data on a public cloud, while also allowing for searchability through the user's encrypted data. Users are able to submit encrypted keyword queries and, through a symmetric searchable encryption scheme, the system finds all files with such keywords contained within.

The system is designed in such a manner that trust from a public cloud provider is not required. The solution satisfies confidentiality of data; data integrity is maintained, file sharing is catered for and a user key-revocation scheme is in place. A further advantage of this 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 also do not need to worry about Cloud Providers gaining access to their data illegally.

The architecture of the system consists of two components, the Client side application and the Server application running on the compute cloud. The client side application performs all the security operations on the data. Along with saving and retrieving data from the Storage Service, the Server application performs the processing involved in handling the encrypted queries. The performance overheads of such a system are potentially significant in terms of additional processing time and the size of the additional meta-data needed.

Preliminary results show that the storage overheads remain fairly constant as input file sizes increase - as file sizes were increased from 3Kb to 147Mb, the security overhead remained between 1038b and 1053b. This overhead is basically insignificant when storing large files.

Overall the benefits of a searchable encrypted cloud service are significant and the approach is viable for using public clouds while still retaining control of the data.

I. Introduction

Cloud computing is a model where various services are offered over the Internet. There are a wide variety of services offered by major companies, from applications to virtual machines to storage space. By using a cloud provider, users need not worry about the underlying implementations and technicalities. For example, if a user has a Virtual Machine (VM) running on Amazon's Elastic Compute Cloud (EC2). The user does not need to worry about the complexities of running a data center. Users interact with the service by using the interface provided.

There are various levels of abstraction in Cloud Computing, namely *Infrastructure as a Service*(IaaS), *Platform as a Service*(PaaS) and *Software as a Service*(SaaS). The difference between these is the amount of control users have over the software stack.

As appealing as the concept is, there are disadvantages, as described by Armburst et al. in [1]. In their work they describe a number of obstacles within the Cloud Computing space, such as the "Data Confidentiality" obstacle. A common sentiment is that companies do not want to store their sensitive corporate data within the cloud.

For this reason it was necessary to investigate a method for providing users with a way of storing their data on a Public Cloud without concern about security breaches at the provider, or the provider accessing their confidential data. Searching through encrypted data was an issue so it was necessary to use current research in *Searchable Cryptography* to allow users to search through their encrypted content.

The paper is structured as follows: Section 2 introduces Related work. Section 3 describes the Requirements of such a system. Section 4 covers the Data Structures used in the system. Section 5 discusses the Architecture of the system and Section 6 will discuss the Preliminary Results gathered by this research. The paper concludes with some analysis in Section 7 and propose some future work ideas in Section 8.

II. Related Work

There are a large number systems that have been designed to secure distributed storage. These storage systems that don't trust the storage medium are the most relevant in our work. Systems like Sirius [2], Secure Network-Attached Disks [3] and Plutus [4] all secure the data at the client before sending it to the server. This technique allows data to be stored on untrusted servers and so, even if there is a

TABLE I
SYSTEM REQUIREMENTS

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

security breach at the server, the data will not be compromised.

Searchable encryption is the idea that a user can encrypt data and then search for keywords within that data later to find possible matches. Our work uses the techniques used by Waters et al. [5]. They developed two systems that use both symmetric and asymmetric searchable encryption techniques. Their work is in turn based on the work done by Boneh et al in [6].
Waters et al. [5] develop a system using two different types of searchable schemes. The first is a symmetric scheme and the second is an asymmetric scheme. They state that a major drawback of the symmetric scheme is that an adversary may compromise the secret key stored by the server.

Major Internet companies have found it commercially viable to sell unused computing resources to the public. In order to achieve this these companies have had to develop systems that can handle very high system loads and that can scale across hundreds of thousands of servers. Companies such a Google, Facebook and Amazon have published a few papers on how they achieve scalable storage. Google uses the Google File System(GFS) [7] to store its large amounts of data. Many higher level services such as Big Table [8] run on top of GFS. Facebook uses Cassandra [9] for their mailbox storage system. Amazon uses Dynamo [10] for their key-value store as used by many of their services. A lot of these systems use Consistent hashing [11] or some variant to achieve scalablility.

III. SYSTEM REQUIREMENTS

Before designing the system it was necessary to consider the security requirements for this system. The security requirements are displayed in Table I

The system needs to ensure that the data being stored remains confidential; this means that only authorised users are able to gain access to encrypted content. The system needs to ensure that the integrity of a file is maintained and to detect any unauthorized changes. As with any file system, there has to be mechanisms in place to ensure that files can be shared amongst users. With the ability to provide users with access rights, the system should also be able to remove user rights. Filtering through encrypted data is an issue, so it was necessary to provide users with the ability to search through encrypted content and return results matching such queries. The system should also be able to recover from a compromised key pair. The system must ensure that only valid client applications are able to connect to the server.

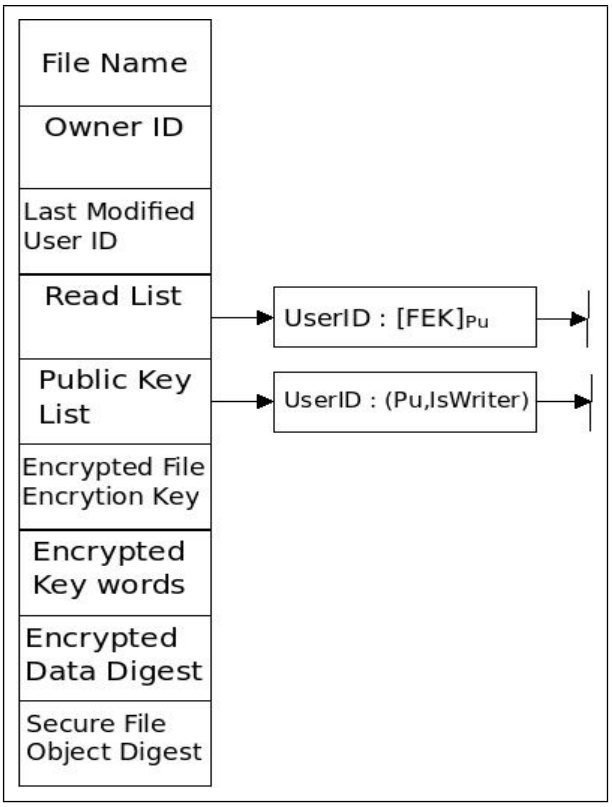

Fig. 1. Secure File Object Data Structure

IV. SYSTEM DATA STRUCTURES

A. Secure File Objects

The *Secure File Object* (SFO) is the container that is used to securely store data on a Storage Service. All operations performed on this container are handled on the client; any unauthorized changes or modifications on this container will be detected at the client.
In order to fulfill the requirements specified in Section III the SFO needs to have a number of meta-data fields. The Fields needed are shown in Figure 1.

Each user of this system needs a Public/Private Key-pair as this allows for non-repudiation.
Data and SFO integrity is maintained by the *Encrypted Data* and *Secure File Object* digests. The Secure File Object digest can only be created by the file owner, whilst the Encrypted Data digest is created by any user that has write access. The SFO digest is a signed hash(Signed with the owners Private Key) of all the fields to which only the owner can modify. The Encrypted data digest is simply a signed hash

of the encrypted data field, signed by the *Last Modified Users* Private key. The encrypted keywords field is the list of keywords encrypted using a symmetric encryption algorithm. Its use is further explained in Section V-E

User rights are maintained using the two lists, namely the Read List and the Public Key List. The read list is simply a mapping of a User ID to an encrypted *File Encryption Key* (FEK). The FEK is encrypted with the user's Public Key - this allows the user to retrieve the FEK by looking up the User ID from the list and decrypting with the corresponding Private Key.

The *Public key* list is used for three operations: Integrity Checks, Write Access and Key-Revocation. The Public Key list maps User IDs to a [Public Key,IsWriter] tuple. When a user modifies the encrypted content, that user generates an encrypted data digest which is signed with the user's private key. The user then sets the *Last Modified User ID* field with his own User ID. Subsequent users can then look up the Last Modified Users ID Public Key in the Public Key list and use that to check the integrity of the Encrypted Data digest.
The Public key list is used to maintain a list of users with write access rights. This is done by setting the *IsWrite* field. If a user's write flag is set, this means that a user has rights to modify the content. If an unauthorized user modifies content then this transgression will be detected by subsequent users.
Key-Revocation is achieved by generating a new FEK and using the Public Keys stored in the Public Key list to generate a new *Read List*.

1) 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 IsWriter flag is not set, the system will generate an error because an unauthorized user has modified the contents.

If a read user adds write rights 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 user permission lists.

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.

B. 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 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 V-E.

The SFO Keywords three fields are as follows:

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

How these three fields are related and used in finding specific keywords is explained in more detail in Section V-E.

C. 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 V-C.

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 V-D.

V. System Architecture

The design of this system had to use techniques from secure distributed storage systems where the storage medium is not trusted. This allowed for all trust to be removed from the Cloud provider.
The system was designed to have two components: a client application that will handle all the security operations, and a server application running within the Compute Cloud[1] that will interact with the Storage Service[2] as shown in Figure 2. The server application will also provide users with the ability to search for keywords.

A. Client Application

As mentioned earlier, the client application is responsible for all the cryptographic operations that will be performed on the SFO. The user logs into the application providing an RSA Keypair as specified in a settings file. The Settings file also includes the address of the Compute Cloud Server Instance, the ID of the User Logged in, the Alias within the keystore and Storage Access keys.
Once a user is logged in they can perform a number of functions based on their access rights. A user with read rights can perform the following operations:

- Load an SFO from the Storage Service
- Check the SFO and Data integrity
- List Users and Rights
- Save an SFO to the Storage Service
- Save the SFO contents to the File System.

The only operation that may seem peculiar is "Save an SFO to the Storage Service". These users can perform this operation since they are not allowed to change the contents, so in essence they are simply uploading the same SFO that they have downloaded. As mentioned in Section IV-A1, malicious users with only read access can modify file contents but such actions will be detected because these users cannot generate a contents digest.

Users with write access rights can perform all the operations of the read users along with the following operations:

- Modify Contents
- Generate Encrypted Data Digest
- Set Last Modified User ID

As explained in Section IV-A, users with write access need to be able to generate the Encrypted Data Digest and set the Last Modified User ID so that subsequent users can validate the authenticity of the changes performed on the contents.

The file owner has rights to perform all the operations of the previous two users along with the following operations:

- Create SFO
- Add Users
- Modify User Rights
- Remove Users
- Generate SFO Digest
- Generate Secure Keyword list
- View / Modify Keywords
- Generate Search Capability

Only the file owner is allowed to modify the keywords attached to a file. The details of these operations is explained in Section V-E.

The Client application executes a number of file system commands namely:

- Put
- Get
- Delete
- List
- Search

The *Put* command is used to store an SFO on the Storage Service. The *Get* command is used to retrieve a file from the Storage Service given a file name. The *Delete* command is used to delete an object from the Storage Service given a file name. The *List* command is used to return a list of items within the Storage Service[3].The *Search* command takes a search capability and sends this request to the server; the server then responds with a list of files matching the search capability as explained in Section V-E.

B. Server Application

The Server Application is used to handle all client requests and runs in the Compute Cloud. This application is used to authenticate clients and provide a secure connection between the client and the Storage Service. The bulk of the processing that is performed is when responding to search queries as described in Section V-E, otherwise it simply receives requests and pushes them forward to the Storage Service.

C. 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[4]. The server can then validate the user by hashing its Access keys to those received from the client. If the hashes match, the client is allowed to access the system. The server then stores the client's public key and

[1]Such as Amazon's Elastic Compute Cloud
[2]Such as Amazon's Simple Storage Service (S3)
[3]In the case S3, returns a list of objects in the bucket
[4]In the Case of S3 a secret key and access key is needed

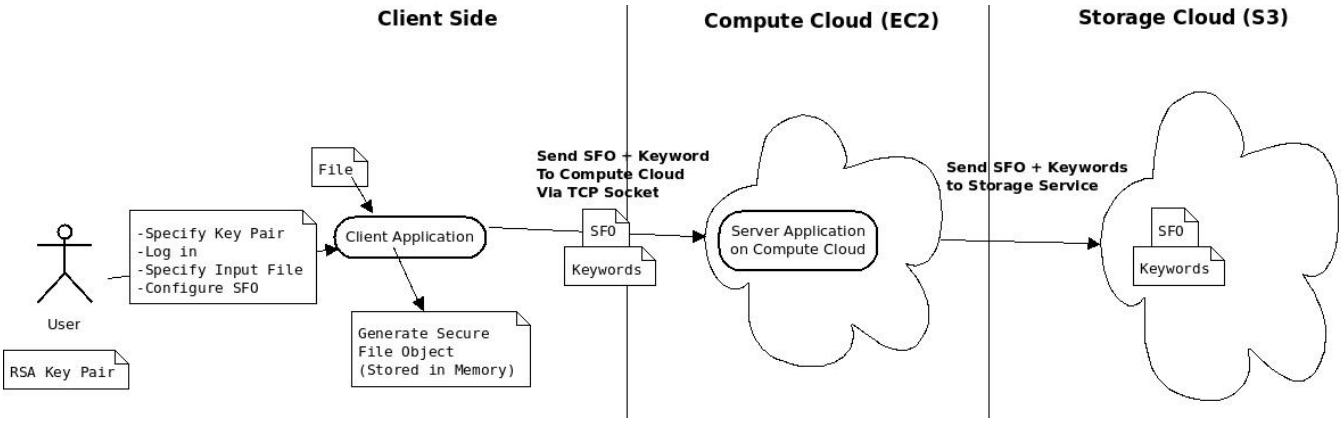

Fig. 2. Overall System Architecture

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 V-D.

D. File-System Operations

Once a secure connection has been established between the client and the server as explained in the Section V-C, the client can then send file 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:

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}]

2) Download Commands: Such as 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}]

E. Searchability

This is the operation when users can submit encrypted keywords to the server. The server then performs cryptographic operations over the files in the Storage Service, returning the results of the query. The scheme used by the server to determine the results of the query is derived from the Searchable Symmetric scheme that was developed in [5]. Waters et al. [5] 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 this system. 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 an encrypted keywords field attached to each SFO. This field is encrypted using a symmetric algorithm with the file owner's secret key so that only the owner may see what searchable keywords are attached to a specific SFO. This extra field in

the SFO plays no part in the query process though.

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 Keyords file Section IV-B. 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 Storage Server.

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 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 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.

VI. Proof of Concept Preliminary Results

In developing a secure storage system, it is necessary to keep the security overhead low with regards to performance and storage. A user would not want to wait for long periods of time while the system is busy performing all the cryptographic operations. Since this system is storing data on a Storage Provider, one does not want to have a large security storage overhead since this will incur higher usage costs.

Figure 3 shows a graph of the security overhead for varying file sizes for AES and DES encryption algorithms. This experiment was set up to only have one user per file and only four secure keywords of five characters in length per file. All the filenames were six characters in length. It is important to keep these attributes static as they can distort the security overhead sizes of the various files.

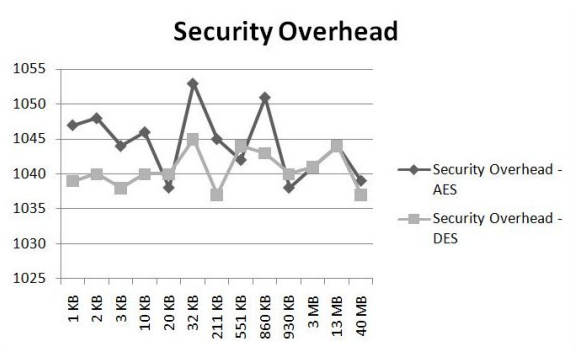

Fig. 3. Additional Security Storage Overhead

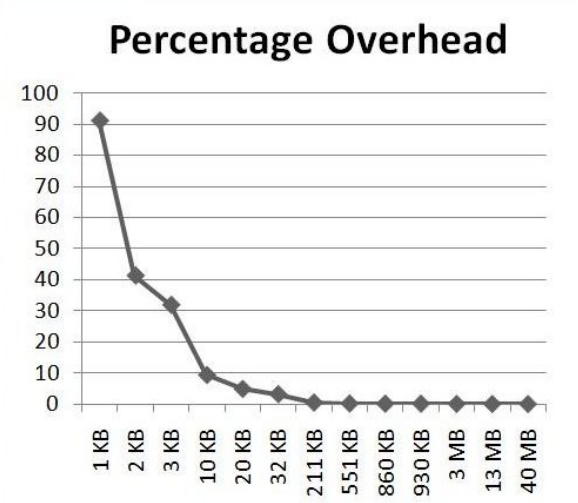

Fig. 4. Additional Security Storage Overhead as a Percentage of Input Size

The total overhead is the overhead of the SFO as well as the overhead of the Secure File Object Keywords file. The size of the *flag* and *random bit string* depends on input parameters. In this experiment the sizes of each were 17b and 65b respectively. The size of the encrypted keywords field for four keywords was 80b which equates to 162b of additional secure keywords storage per Secure File Object.

The file sizes range from approximately 3Kb to 147Mb. As is shown in the graph the size of the security overhead is between 1038b and 1053b.

VII. Analysis

Preliminary results show that this system places very little overhead on files. In doing an analysis of the Secure File Object data structure, the additional storage needed comprises of a static amount plus a variable amount that is dependant on the number of read and write users and the number of

keywords, as shown below:

Overhead = 566b + 136b(r) + 136b(w) +8b(w)

where b is *bytes* and 556b is comprised of the new file name and the owner id**ADD** (however these size changes are negligible). According to [12] the average length of an English word is 4.5 letters, rounded up to 5 letters, and equates to approximately 8 bytes of encrypted text hence the 8b(w).

As shown in Figure 3, using a DES encryption scheme is more efficient in terms of storage overhead than an AES encryption scheme. The overhead formula does not take into account the increase in size of the encrypted content, meaning how much more space does the encrypted data occupy over the unencrypted data. On average this was approximately 4 Bytes more with a Standard Deviation of 2.47. The spikes in the graph are due to the additional padding needed by the DES and AES encryption algorithms. These spikes are never more than one block; for DES this is an additional 8 bytes and 16 bytes for AES. It is for this reason that the AES graph has larger spikes than the DES graph.

Figure 4 illustrates the percentage overhead of security for a given input size. The graph shows that as the size of the input file increases, the percentage overhead drops. This is due to the security overhead remaining fairly constant, regardless of the input size. For files of 10Kb in size, the overhead is approximately 10%, then as the file sizes increase over 200Kb this overhead drops to below 1%.

VIII. Conclusion and Future Work

This paper has shown that Searchable Secure Storage can be achieved on a Public Cloud provider with minimal storage overheads. Users are able to store their data securely on an untrusted Cloud Storage service along with the ability to search through their data. The architecture follows a traditional Client / Server model with the client performing all the cryptographic operations and the server performing the search operations in a secure and confidential manner. Preliminary results show that the security overhead with regards to space is minimal. As file sizes increase, the percentage overhead becomes less than 1% for file sizes greater than approximately 200kb. The processing overhead for this system is currently being investigated.
Further research includes exploring the use of encrypted indexes in improving the processing time needed to respond to user queries.

References

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

[2] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*. Citeseer, 2003, pp. 131–145.

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

[4] 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*. USENIX Association, 2003, pp. 29–42.

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

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

[7] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[10] 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*, vol. 41, no. 6, pp. 205–220, 2007.

[11] 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*. ACM, 1997, pp. 654–663.

[12] C. Shannon, "Prediction and entropy of printed English," *Bell System Technical Journal*, vol. 30, no. 1, pp. 50–64, 1951.