A Middleware Approach for Building Secure Network Drives over Untrusted Internet Data Storage

Ravi Chandra Jammalamadaka†, Roberto Gamboni*, Sharad Mehrotra†, Kent E. Seamons‡,
Nalini Venkatasubramanian†
University of California, Irvine†, Brigham Young University‡, University of Bologna, Italy*
{rjammala, sharad, nalini}@ics.uci.edu seamons@cs.byu.edu roberto.gamboni@studio.unibo.it

Abstract

In this paper, we present the design of DataGuard middleware that allows users to outsource their file systems to heterogeneous data storage providers available on the Internet. Examples of data storage providers include gmail.com, rapidshare.de and Amazon S3 service. In the DataGuard architecture, data storage providers are untrusted. Therefore, DataGuard preserves data confidentiality and integrity of outsourced information by using cryptographic techniques. DataGuard effectively builds a secure network drive *on top of any data storage provider on the Internet. We propose techniques that realize a* secure file system *over the heterogeneous data models offered by the diverse storage providers. To evaluate the practicality of DataGuard, we implemented a version of the middleware layer to test its performance, much to our satisfaction.*

1 Introduction

Recently, there has been an explosion in the number of Internet data storage providers (IDP) that are emerging. Examples of such services include: Rapidshare.de, Youtube.com, Megaupload.com, Yahoo Briefcase!, Amazon S3 service, etc. The clients outsource their data to IDPs, who provides data management tasks such as storage, access, backup, recovery, etc. IDPs offer numerous benefits to users, which include: a) *Device Independence:* Clients can access their information from any machine connected to the Internet; and b) *Data Sharing:* The IDPs provide data sharing capabilities that allow users to share their data with any user on the Internet.

Currently, users employ a variety of ways to achieve mobility when it comes to personal data. The range of solutions include but are not limited to: i) Carrying their data in secondary storage devices such as USB drives, CD/DVDs, etc. This is a largely inconvenient solution pushing the burden of data management to the user. Also, the solution in inherently insecure, as most users store their data in plaintext. Such devices can be easily lost or stolen; ii) Maintaining public servers such as web servers, FTP servers, etc. The drawbacks of this solution are twofold: a) Administering such a service is burdensome and requires sound technical knowledge; and b) Many users are not in a position to run such a service due to ISP restrictions. Likewise, to share data, users employ solutions like sending email, etc., which suffer from similar drawbacks listed above.

By comparison, services offered by the IDPs do not suffer from the above drawbacks and have the following advantages: a) *Availability:* Data is available 24/7 from any computer connected to the Internet; b) *Low cost:* Typically, the services are free. The business model is based on advertisements, emphasizing the fact that storage has become very cheap; c) *Good Service:* The storage providers typically employ experts, thereby providing very high quality service. All of the above advantages make IDPs an attractive prospect for data storage.

The primary limitation of such services is the requirement to *trust* the storage provider. The client's data *is stored in plaintext* and therefore is susceptible to the following attacks:

- **Outsider attacks:** There is always a possibility of Internet thieves/hackers breaking into the storage provider's system and stealing or corrupting the user's data.
- **Insider attacks:** Malicious employees of the storage provider can steal the data themselves and profit from it. There is no guarantee that the confidentiality and integrity of the user's data are preserved at the server side. Recent reports indicate that the majority of attacks are insider attacks [9, 8].

Despite these security concerns, IDPs are gaining popularity due to the convenience and usefulness of the data services they offer. In this paper we present the design

and implementation of DataGuard, a middleware based architecture that allows the users to outsource their data to untrusted IDPs. Our goal is to develop a middleware that a client can run on their local machines, which can interact with the IDPs of their choice and yet manage the client's data securely. We address the problem at the file level, i.e., the users outsource their local file system to the IDPs. DataGuard effectively builds a network drive on top of data storage provided by the IDP.

There are three primary reasons that motivates our work on designing such a middleware: a) **Popularity:** Network drives are very popular because they allow users remote access to their data. They effectively provide a virtual disk that users can carry around seamlessly without much effort. This is precisely the reason why there are many commercial IDPs offering a *network drive like* service on the Internet [22, 23]; b) **Security:** Data should be secured before being outsourced to an untrusted server. b) **General applicability:** A wide variety of applications can be supported by a *file storage* like service. For instance, consider the following sample applications that can be supported: a) an *autofill application*, which remembers and fills out passwords from any machine connected to the Internet. b) a *bookmark manager* which provides remote access to personal bookmarks.

DataGuard allows users to specify which IDP they want to store their data. To provide such functionality, DataGuard needs to take into account the heterogeneity of the data models that are offered by the IDPs. For instance, in Amazon S3 service, files are the basic units of data, while in Gmail.com, emails are the basic data units. One of the fundamental tenets of DataGuard is make sure that *no changes are required at the server to support DataGuard*. The servers are oblivious to the existence of DataGuard. To combat such heterogeneity, DataGuard provides a novel general model of a file/data, that can then be further customized to individual IDPs. We will call this model as the *generic data model* (GDM). We will propose techniques to map the generic database model to server side data representation. Since the IDPs are untrusted in our model, we propose a *security model* that will allows DataGuard to ensure data confidentiality and integrity of user's data by using cryptographic techniques.

DataGuard supports all the operations supported by modern file systems such as creating a directory, reading a file, etc. DataGuard also allows users to search for documents that contain a particular keyword. Such a task in context of DataGuard is very challenging, since the data is encrypted at the server. The obvious solution of fetching all the encrypted data from the server, decrypting it and executing the query locally is impractical as it puts tremendous performance strain on the system. We develop a novel index based approach of executing such keyword based queries at the server. The proposed index is carefully designed not to disclose any information to adversaries. Previous work [7, 4] on executing queries over encrypted data cannot be utilized in the context of DataGuard, since the previous work assumes that the server is cooperative and runs a compliant protocol for enabling search. We cannot make such an assumption, since in the DataGuard architecture, no changes are possible at the server.

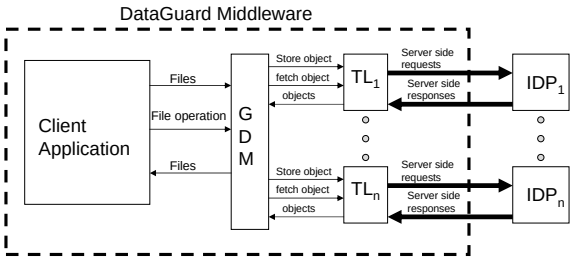

Figure 1. DataGuard Architecture

Our contributions in this paper are the following: a) A novel middleware based architecture called DataGuard, that allows users to outsource their information to any Internet data storage providers of their choice; b) A novel data model called the generic database model that can be easily translated to the specific data models at the server; c) A security model that allows DataGuard to enforce security constraints of the user at the data level; d) A novel index based approach of executing such keyword based queries at the server; and d) A prototype implementation of the DataGuard middleware.

RoadMap: In section 1.1, we provide a brief overview of the DataGuard architecture. Section 2 presents the generic data model. In section 3.1 and 3.2, we present our security model over GDM. In section 4, we present the different client-server interactions in our model. In section 5, we describe the specifics of the translational layers. In section 8, we present our novel cryptographic index that handles keyword searches on encrypted data. In section 7, we present the performance results of the DataGuard prototype.

1.1 Architectural Overview

The desirable properties/goals of DataGuard are the following:

- Allow users to outsource their file system to any Internet based data storage provider of their choice.
- Preserve security properties of user data such as data integrity and confidentiality.
- DataGuard should be easy to use.

Major Entities and Threat Model: There are three main components in our architecture: a) Client machine;

b) DataGuard middleware; and c) Data storage providers. The Client machine is the end device from which the user is accessing the data. The client machine is entirely trusted. The DataGuard middleware and it's associated components, i.e. *translational layers* are also trusted and run inside the client machine. We will explain the role of the translational layers soon. The middleware is in charge of providing data services to the user by fetching the required data from the storage providers. The storage providers provide data management services to the clients and are untrusted. We will assume a *honest-but-curious* behavioral model for the storage providers. That is, the storage providers are expected to provide the required services, but the employees that work for such providers could steal data and profit from it.

Overview: Figure 1 illustrates the overall architecture of DataGuard. DataGuard is both a client application and a middleware that runs at the user's client machine. When the application is first started, the middleware will ask for the name/URL of the storage provider, username and password for authenticating to the storage provider. In addition, DataGuard requires users to provide a masterpassword, a secret used to enforce data confidentiality and integrity. Masterpassword is the only secret that the user needs to remember for using DataGuard and it is used to generate all the cryptographic keys. After the user specifies the data in a login screen, the middleware fetches the file system content from the server and provides a *file system like* view to the user. All the standard operations of a regular file system are available to the user. Once the user closes a session, any temporary files that are opened by the middleware application are closed. The user can repeat the process for multiple different storage providers and DataGuard provides a common interface for accessing all the respective file systems.

The atomic unit for data retrieval in DataGuard is *files*. The client application requests files from the middleware. But DataGuard middleware does not work with a file based data model. Instead, it works on an object based data model called the generic data model (GDM). DataGuard middleware first maps files to objects in GDM and also translates file system operations to their equivalent operations on the object. We will discuss them in more detail in section 4. The object level abstraction is necessary since IDPs vary significantly when it comes to data storage models. For instance, Yahoo mail and Gmail work with email based data abstractions, and Rapidshare.de and MegaUpload.com work with a file based data abstractions. The objects are then further translated into individual data models of the storage providers by *translational layers*. Translation layers contain functions that store and fetch objects from the IDP. Since the implementation of these functions vary from IDP to IDP, translation layers need to be written for each individual IDP. We envision that these translation layers will be written by experts or the webmasters of the IDPs themselves, if DataGuard becomes popular. Currently, we have written three such layers which we will introduce soon. The translational layers can be written fairly easily for most IDPs on the Internet.

Objects provide a higher level/general enough abstraction, that allow data items smaller than a file to be stored and fetched from the server. Although, in DataGuard's current avatar we do not fetch anything smaller than a file. We adapted the object oriented approach to accommodate our future directions with regard to DataGuard. Alternatively, instead of operating at an object based data model, DataGuard could have followed an XML based data model. The drawback of such an approach is: XML requires fairly thick parsers that make the DataGuard middleware fairly bulky. We therefore, decided to design a fairly simple but flexible enough object based data model called the generic data model that we will describe in the next section.

Our goal is to develop DataGuard and release an API that will allow experts or webmasters to release translation layers based on this API that will allow users to use a variety of IDPs as storage servers.

2 Generic Data Model

The *generic data model* (GDM) is an object-based model for representing files and directories in DataGuard. The GDM middleware creates object instances and invokes operations on them according to an API that supports the storage and retrieval of DataGuard objects. The GDM is simple and generic enough that to support a variety of storage providers.

Each GDM object O has a unique id ($O.id$) and a set of attributes $O.A$ where $A = \{id, content, metadata\}$. $O.content$ represents the object's content and $O.metadata$ represents the ancillary information about the object. The metadata is a set of *attribute=value* pairs. DataGuard stores and deletes data at the object level at the storage provider. Updates to objects are modeled as a delete operation followed by an insert/store operation.

2.1 Mapping a file system to the GDM

Conceptually, a file system F_S can be represented as a graph having the structure illustrated in fig 2. Every directory and file is a node in the graph. An edge between two nodes represents the *parent-child* relationship. The "*" operator implies zero or more nodes, as a *directory* node can have zero or more files or sub-directories.

In DataGuard, every file and directory node is treated as an object. Each object has its own *unique id*. We will explain the generation of such ids shortly. For a file, the

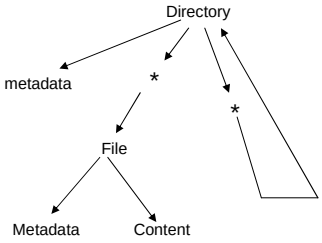

Figure 2. Graph representation of a file system

object's content ($O.content$) is the content of the file. The object's metadata ($O.metadata$) includes information such as the file name, last modified date, file size, etc. The object's name ($O.name$), is the name of the file.

For a directory, the object's content ($O.content$) is set to *null*[1]. The object's name ($O.name$) is the name of the directory. The metadata of the directory object will include directory name, size of the directory, directory contents, etc.

Additionally, the directory object maintains a *child_references* attribute in the metadata. Child_references contains a list of pointers to the immediate children of the directory node/object. The pointers contain the id of the object being referenced to allow DataGuard to fetch the child nodes of a directory when required.

Id Generation: The object ids are randomly generated. For every object O_i, a ramdom number r_i is generated and hashed deterministically $h_{MP}(r_i)$ using the master-password (MP) as the key to determine the object's id $O_i.id$. The reason for hashing the id will be apparent in the next section.

3 Enforcing Security Constraints

3.1 Data Confidentiality

In DataGuard, the user's outsourced data is kept confidential. A DataGuard object *0* contains the following attributes { id, name, content, metadata }. This section describes how the confidentiality of DataGuard data is achieved.

id attribute: The id of the object does not reveal any information about the object at the server side and hence it is left untouched. The object's id is used to fetch it from the server.

Content, name and metadata attributes: An object's metadata, name and content are encrypted using the *object's encryption key* (OEK). The key is generated on the fly using the key derivation function (KDF) of the password-based encryption specification PKCS #5 [5]. The KDF function calculates keys from passwords in the following manner:

$$Key = KDF(Password, Salt, Iteration)$$

The *Salt* is a random string to prevent an attacker from simply precalculating keys for the most common passwords. The KDF function internally utilizes a hash function that computes the final key. To deter an attacker from launching a dictionary attack, the hash function is applied repeatedly on the output *Iteration* times. This ensures that for every attempt in a dictionary attack, the adversary has to spend a significant amount of time. In DataGuard, to generate the OEK for an object O, we use the masterpassword as the password, the id of the object as the salt, and we set the iteration count to 1000, the recommended number.

Another approach is to generate a random key for each object and encrypt the object with that key. The random key could then be encrypted with the key derived from the masterpassword. We chose to generate the key since it is inexpensive using a hash function rather than retrieve a key along with each object from the server. This saves network overhead, especially for small objects when the cost of retrieving the key would dominate.

3.2 Data Integrity

Another requirement of DataGuard is that data integrity be preserved. This section describes how DataGuard ensures that data is both *Sound* and *Complete*.

Soundness: To ensure soundness of an object, DataGuard needs a mechanism to detect when tampering occurs. To achieve this, the HMAC[2] of an object is calculated and stored on the server. When the object is retrieved from the server, its HMAC is also returned. The client calculates an HMAC again and compares it to the original HMAC. If they are equivalent, then no tampering has occurred. One way to compute the HMAC is as follows:

$$HMAC(O.id||O.name||O.Content||O.metadata)$$

Although the HMAC can be used to determine soundness, it does not guarantee the *freshness* of the object. That is, the server could return an older version of the object and the client will fail to detect it. One way to address this is to include the current version of the object when generating the HMAC. Thus, the HMAC can be generated as follows:

$$HMAC(O.id||O.name||O.Content||O.metadata||Version)$$

Every time the object is updated, the version number is increased and the HMAC calculated again. This is done at

[1] The directories in many modern file systems do not actually carry any user data.

[2] A keyed-hash message authentication code.

the client side and hence there is no loss of security. Another possibility is to use the last modified date of the object as the version number. Such a date could be stored in the object's metadata. When access to the object is desired, the object can be retrieved from the server and the HMAC calculated again locally. The client now has to confirm the version number or the last modified date manually to determine if any tampering as taken place. For instance, if the client does not agree with the last modified date/version that computes the HMAC the server returned, then the client can detect that a possible tampering has taken place at the server. But such an approach requires the user to validate every object and hence makes the system unusable.

Another method is to calculate the *global signature* of the complete file system using a Merkle tree approach [**?**] and store the signature locally. Whenever access to an object is made, the server sends a partial signature over the remaining objects so that the client can use the partial signature and the object being accessed to generate a signature to compare to the most recent global signature that is stored locally. If the signatures match, then no tampering has occurred. We did not adopt this solution because it requires server-side support, and violates our goal to use existing data storage providers. Also, it requires a mobile user to transfer the global signature between machines, thereby pushing data management tasks back to the user, something that we want to avoid. An open problem is to design data integrity techniques that allow the client application to detect data tampering attempts at the server, without any user involvement.

DataGuard leverages the ability to store data with multiple storage providers. For instance, if a user configures DataGuard with at least two different storage providers SP1 and SP2, then DataGuard can store the version numbers of the objects belonging to SP1 with SP2 and vice versa. If we assume that SP1 and SP2 do not collude then the last update problem can be solved. This assumption of non-colluding servers has been made previously by the authors of [3] and it is applicable to DataGuard since the storage providers are in two different administrative domains to reduce the probability of an attack.

Completeness: In DataGuard, the completeness property needs to be verified when fetching all the child nodes of a directory. The information about the of number of children of a directory node is stored in the metadata of the object (in the child_references attribute). This information lets the client know precisely how many child objects should be available from the server.

3.3 Hiding File Structure

The file system-GDM mapping discussed previously translates a hierarchal representation of a file system to a flat structure (i.e., object representation). The objects are encrypted before being stored at the server. The benefit of such an approach is that it hides the structure of the file system at the server. In [24], the authors have identified the benefits of hiding the file structure from the untrusted server.

Notice that mere encryption of the objects does not completely hide the structure of the file system. When the metadata of the object is encrypted, the number of pointers in the child_references dictate the amount of the ciphertext of the object. The size of the ciphertext leaks the structure of the file structure, although not completely. The adversary can now determine the number of the children of a directory node from the ciphertext size, but he/she does not know where the actual child nodes are stored.

To prevent even the partial structure leakage, all the directory objects need to be made of equal size. This is done as follows: DataGuard preordains a number called MAX_CHILDREN. For directory nodes that have child nodes less than MAX_CHILDREN, DataGuard pads the *child_references* attribute with placeholder pointers to make size equal to the size of MAX_CHILDREN references. If a directory node contains more than MAX_CHILDREN child nodes, DataGuard splits them into a set of directory objects each containing MAX_CHILDREN nodes. We will explain the details of the process with an example: Let an object O contain n children, where $MAX\_CHIDREN < n < 2{*}MAX\_CHILDREN$. Now DataGuard splits the object O into two object O_1 and O_2. The id of O_1 will be same as that of O. The id of object O_2 is calculated as follows:

$$O_2.id = h_{MP}(O_1.id||Index)$$

$||$ represents the string concatenation symbol. *Index* is an integer whose value is incremented for each additional object that is created. In the above example, *Index* value is set to 1. The hashing is done to ensure that DataGuard with the knowledge of $O_1.id$ can calculate $O_2.id$. MAX_CHILDREN references/pointers of nodes from the n children of O are stored in 0_1 and remaining $n - MAX\_CHILDREN$ references are stored in O_2's $Child_{References}$ attribute. The O_2 Child_References is padded with placeholder pointers to reflect the size of MAX_CHILDREN pointers. The object O_1 inherits the metadata from the object O and metadata of the object O_2 is set to null [3]. The O_1 object's metadata maintains an *additional_objects* attribute which tells DataGuard the number of additional objects its needs to fetch to complete the directory object. We will refer to 0_1, the first object that is fetched as the *starter object* and the rest of the objects as the *additional objects*.

[3] In reality, the metadata of this object is also padded up with placeholder bits to ensure equal size between the objects.

Now when the server has to fetch object O from the server. It first fetches the starter object O_1. Then it utilizes the value of the additional_objects attribute to obtain the ids of the all the subsequent objects by hashing the id of O_1 together with the index counter that is incremented for each additional object. DataGuard now has enough information to recreate the object O at the client side. Notice that that the hash function is used to calculate the id of the additional objects from the id of the starter object. To have a uniform representation for object ids, the random number generated for the starter object is also hashed.

Our storage model prevents an adversary from determining the file structure from the ciphertext. However, a sophisticated adversary can still infer the structure from the access patterns. A solution to this problem will require a solution similar in spirit to oblivious computing [**?**, **?**]. Such a solution will be computationally very expensive and impractical in the DataGuard scenario. We designed a solution that strikes a balance between security and performance.

4 GDM Operations

This section describes the interface/functionality provided by the middleware. The interface is a set of functions that translate file system operations to operations on the server/storage providers side.

Login (storageproviderURL, username, password, masterPassword): The middleware procures the URL of the storage provider, username and password of the user to be able to access the service provided by the storage provider. Once the user provides the right credentials, DataGuard fetches the root object from the server and displays the object as a directory to the user. In DataGuard the id of the root object is equal to $h_{MP}(1.0)$, where MP represents the masterpassword of the user. The user can now perform any of the following standard file system operations.

Create_File (fileName): A new object is created to represent the file. The object id is randomly generated (see section 3.1). Then, object's metadata and the content are encrypted and the data signature is calculated as described earlier in section 3.1 and 3.2. The object is then stored at the server using the *store(Object 0)* function implemented in the translation layer of the IDP. The directory object's metadata under which the file is being created is also fetched. The children_refereneces attribute is then manipulated to store the pointer to the newly created file. The parent object is then stored at the server.

Open_Directory (directoryName): The middleware identifies the children of the directory node using the child_references attribute. The objects are then fetched from the server and presented in the form of directories and files to the user. Here, the object's metadata only is fetched.

Read_File (fileName): To read/open a file, the user first navigates to the file and clicks it. The middleware then procures the object id and fetches its content from the server.

Write_File (fileName): When updating a file, the data signature is calculated as described earlier and the object's content is encrypted using the OEK that is generated using the id of the node/file. The old object residing at the server is deleted and the new version of the object calculated in the previous step is then stored at the server.

Move_File (fileName, destinationPath): Let us assume a file f is moved from directory d_1 to d_2. In DataGuard the move operation is enforced by removing the pointer to f from d_1 and adding the pointer to f in d_2. Even for this operation, it is only required to fetch the metadata of the objects d_1 and d_2. The ids of the objects are not changed during the move and hence they do not effect the cryptographic keys that secure the object.

Move_Directory (directoryName, destinationPath): This operation is similar to the Move_File operation discussed above.

Delete_File/Directory (file/directory_Name) : When deleting a file, its corresponding object is deleted from the server. When this is done, the parent object is also fetched from the server and pointer deleted from the child_references attribute.

We have ignored the description of other operations such creating a directory, renaming a directory, etc, due to the lack of space. The previous descriptions of some of the operations provide intuition for the other operations. For more details regarding the client server interactions, please see the full version of the paper [6].

5 Translational Layers

A TL layer contains the implementation of five functions illustrated in fig 3. The function *store(Object O)* stores an object O at the server and the function *fetch(Object O)* retrieves the object from the server. In reality, the fetch function is overloaded with another function *(fetch(Object 0, Metadata)* which fetches only the metadata of the object. The function *delete(Object O)* deletes an object stored at the server. Functions *connect* and *disconnect* open and close a session with the IDP.

```
public int connect(String username, String password)
public int disconnect()
public int store(Object o)
public int fetch(Object o)
public int delete(Object o)
```

Figure 3. Translational layer functions

The translation layer contains the server specific implementation that is required by the DataGuard middleware.

We will now proceed to define two layers supporting email and Amazon S3 storage service systems.

Email translation layer: An email translation layer is useful because: a) Most users have access to web based email services that are free; and b) These services currently provide user's a significant amount of storage space.

Store(Object O:) An object O is mapped to two email messages E_1 and E_2, where E_1 stores the object's metadata $O.metadata$ and E_2 stores the object's content $O.Content$. The metadata and the content are stored as email attachments. The object's id is stored in the subject header of both the email messages. The email is created at the server by generating the appropriate HTTP POST message that creates an email at the server.

Fetch (Object O:) DataGuard fetches the required object by querying the email service provider's search interface to find the appropriate emails. This is done by generating the HTTP POST messages that forwards the search query to the server. DataGuard needs to use the search interface to identify the required emails, as it does not have control over how the emails are stored at the server. Typically, emails are given a server side id. This id is used to fetch emails from the server. DataGuard has no control over the generation of such an id. It needs to use the search interface to procure the id and then use it to fetch the email. Such downloaded emails are then mapped to an object form.

We have described the details of the two of the most important TL functions. In the interest of space, we will not describe the other functions in this paper. We hope the above discussion provides the user with enough intuition for other functions.

Amazon S3 translation layer: Amazon S3 service also follows an object based model for data representation. In Amazon S3 service, data is modeled as a set of buckets. Each bucket contains a set of objects. A bucket cannot contain further buckets inside them. The TL layer for Amazon S3 service maps DataGuard objects to Amazon objects and vice versa.

Store (Object O:) An Amazon object consists of the following primary attributes { Key, DataString }[4]. The DataGuard object's metadata and content are stored in the Amazon object's DataString attribute. The id of the DataGuard object is stored in the Key attribute. An HTTP POST message is generated that creates the required object at the server.

Fetch (Object O:) The TL layer will fetch the Amazon object with *Key* equal to O.id. This is achieved by creating the appropriate HTTP POST message as well.

[4]We have ignored other attributes in the interest in brevity

6 Keyword Search

File systems allow users to search for documents that contain a particular keyword. This section describes how DataGuard handles such queries.

Problem definition: A file f can be represented as a set of words $\{W_1, W_2, \ldots W_n\}$. A keyword search query Q is also a set of words $\{W_1^q, W_2^q \ldots W_m^q\}$. Given a query Q and a set of Files $S_F = \{F_1, F_2 \ldots F_k\}$, our objective is to find all files from S_F, where $\forall i$, $W_i^q \in F_j$, $1 \leq i \leq m$ and $1 \leq j \leq k$.

The problem is relatively easy if the files in S_F are not encrypted, which is not the case in DataGuard. There has been previous work on executing keyword search queries on encrypted data [7, 4]. Solutions previously proposed assumed that the server is cooperative and runs a complaint protocol to enable search on encrypted data. For instance, in [7], the client computes a trapdoor for a keyword and sends the trapdoor to the server. The trapdoor does not reveal any information about the original keyword. The server now by utilizing the trapdoor will perform a linear scan of all the documents to test for ciphertext blocks that contains the required word. Similarly, in [4], the server computes multiple hashes of the trapdoor to compute an index that it can then subsequently use to find the required documents. Clearly, such approaches cannot be applied in DataGuard context, since the server cannot be dictated to perform computations that it is does not perform already. In the DataGuard architecture, we made an assumption that servers can only store and fetch objects and nothing else. The server is not expected to perform any further operations on the objects.

We will now present a novel cryptographic index *CryptInd* that allows DataGuard to search for files which contain the required keywords. *CryptInd* consists of a set of index entries $\{I_1, I_2, \ldots I_n\}$. An index entry contains information about the documents that contain a particular keyword.

BuildIndex: An index entry I_i is a pair $< H_i, E_{Mp}(B_i) >$. H_i represents the hash of a keyword/word and B_i represents a an arrays of bits. We will refer to H_i as the *keyword hash* and B_i as the *bitmap* of the index entry I_i. Let S_f be the set of files being indexed. Every file $f \in S_f$ is given a *document id*. This id is different from the object id we discussed in section 2. The generation of the document ids is done as follows: At the beginning of BuildIndex, a counter is set to zero. For every document being indexed, the counter is incremented and its value is stored along with the file/object as the document id.

Let K_W represent the set of unique keywords that are present in all the file in S_F. For every keyword $K_i \in K_W$, an index entry $I_i =< H_i, E_{Mp}(B_i) >$ is created. The

hash of the keyword $h_{MP}(k_i)$ is stored in H_i, where the masterpassword (MP) is used as the key.

B_i is an array of $2 \times N_d$ bits, where N_d refers to the number of documents initially being indexed. The size of B_i needs to at least N_d bits due to the following: Let $f.id$ represent the document id of a file f. For every file $f \in S_F$, the $f.id$ th bit of B_i is set to 1, if $K_i \in f$. We use the bits to keep track of the documents that contain the keyword. CryptInd contains additional bits for documents that can be created in the future. Therefore, the size of B_i is increased to $2*N_d$, to allow some slack for futures updates. When the number of documents exceed $2*N_d$[5], there are two options: a) build the index again with larger array size; or b) Split the index entry into two. The id of the second index entry is derived by hashing the id of the first entry. A flag bit is set at the end of the first index entry which specifies to DataGuard that an extra index entry needs to be fetched. Building an index can take significant amount of time and hence the latter approach is preferred. Splitting an index entry has one disadvantage: When access to a index entry which has been split is desired, it takes two rounds to fetch both the required entries from the server. In DataGuard, after a significant number of index entries have been split, the index is re-built at the client. The bitmap B_i is then encrypted by using the masterpassword as the key to generate $E_{Mp}(B_i)$. The index entries created in this fashion are then stored at the server.

Storing the index: Each index entry I_i is stored as an object O_i at the server. The keyword hash of index entry i.e., $I_i.H_i$, is stored as the object's name $O_i.name$. *name* is an attribute that is stored in the object's metadata. The encrypted bitmap of the index entry, i.e., $I_i.E_{MP}B_i$ is stored as the object's content $O_i.content$. The metadata of the object $O_i.metadata$ contains $index = true$ attribute-value pair. This will allow DataGuard to identify the object as an index entry. The index objects are given automatically generated ids.

Searching the index: Let Q = $\{W_1^q, W_2^q \dots W_m^q\}$ be a query which is a set of unique keywords. The objective is find all the documents that contain all the words in Q. For every word $W_i^q \in Q$, its hash $h_{MP}(W_i^q)$ is calculated and corresponding index entry is retrieved from the server. Let the set $I^q = \{I_1, I_2, \dots I_m\}$ represent all the index entries retrieved from the server in response to the query Q. All the bitmaps of the index entries in I^q are decrypted. Let $I.B_i$ represent the plaintext/decrypted Bitmap of index entry I. Then, a conjunction of ANDs of all the Bitmaps in I^q, i.e. $\wedge_{j=1}^{m} I_j.B_i$, is calculated. $\wedge_{j=1}^{m} I_j.B_i$ contains information about the documents that satisfy the query. All the files metadata whose ids are equal to the position of the set bits in $\wedge_{j=1}^{m} I_j.B_i$ are fetched from the server. These files contain all the keywords present in the query. The user can now choose to fetch the required file from the server by clicking on it and invoking the *Read_File* function.

Updating the index: For every file f that is updated, CryptInd needs to be changed at the server. Let us assume that a file f is updated to its new state $f^{'}$, DataGuard calculates the differences between f and $f^{'}$. Let W^{add} represent that new words that are added into $f^{'}$ and let W^{del} represent the deleted words from f. Note, care should be taken to make sure that all the words in W^{del} are not present in f elsewhere. All the index entries who keyword hashes correspond to hash of words in the set $\{W^{add} \cup W^{del}\}$ are fetched from the server. The index entries are updated to reflect the changes done to the file f. For all the words in W^{del}, the f.id th bit of corresponding index entry is set to 0. Similarly, for all the words in W^{add} the f.id th bit of corresponding index entry is set to 1.

Security Analysis: In [4], the authors propose a security model for cryptographic indexes. We will use the security model to show that CryptInd is secure. The intuition behind the security model is as follows: *A cryptographic index is secure, if it does not reveal any information about the plaintext/original data*. More formally, the cryptographic indexes need to be *semantically secure*. Semantic security is a strong notion of security, which can be summarized in the context of a cryptographic index as follows:

Semantic security for indexes: Let I_1 and I_2 be two index entries of an cryptographic Index $\mathcal{I}$, corresponding to two different keywords K_1 and K_2 respectively. An adversary $\mathcal{A}$ is provided with $\{K_1, K_2\}$ and $\{I_1, I_2\}$. $\mathcal{A}$ does not know the relationship between keywords and the index entries. $\mathcal{A}$ will try and guess the relationship between them. Let $Pr[I_1 \rightarrow K_1]$ be the probability that $\mathcal{A}$ correctly guesses that I_1 is the index entry for keyword K_1. Note, if $\mathcal{A}$ can correctly decide the relationship between I_1 and K_1, then automatically $\mathcal{A}$ can deduce the fact that I_2 is the index entry for K_2. For the index $\mathcal{I}$ to be semantically secure, the following inequality should hold:

$$Pr[I_1 \rightarrow K_1] \leq (\frac{1}{2} + \epsilon)$$

where ϵ is a negligible real number.

Claim 6.1 *CryptInd is semantically secure*

Proof Sketch: *Consider an Adversary $\mathcal{A}$ with knowledge of two index entries $\{I_1, I_2\}$ and the corresponding $\{K_1, K_2\}$. Without loss of generality, let us assume that A is trying to guess the index entry of K_1. Let $I_1.H_1$ and $I_2.H_2$ be the* keyword hashes *of index entries I_1 and I_2. Both the keyword hashes are of equal length. Recall that a secure cryptographic hash function is used in calculating the keyword hashes of the index entries. Therefore, since secure cryptographic hash functions are* semantically secure, *the adversary by looking at the keyword hashes of the index*

[5]Notice that we are not limited to the scaling factor of two.

entries alone, will not be able to predict accurately the index entry of keyword K_1. Now consider the encrypted bitmaps of the index entries. Standard encryptions functions are also semantically secure, the bitmaps also will not reveal any information. Both the keyword hashes and the bitmaps of the index entries do not reveal any information for $\mathcal{A}$. $\mathcal{A}$ can now only guess randomly to find the correct index entry for keyword k_1 and will succeed with a probability of $\frac{1}{2}$.

In the current version, CryptInd cannot handle pattern based keyword queries, such as *Secur*, which are allowed by the modern file systems. The next section describes how CryptInd can be extended to handle pattern queries. We will refer to the extended index as the CryptInd++.

6.1 Support for pattern queries

Definition q-gram: Let s be a string of length l. A q-gram of a string s, is a substring of s of length q, where $q \leq l$.

Consider the string *secure*. Substrings *se* and *re* are examples of 2-grams of *secure*. Likewise, substrings *sec* and *cur* are examples of 3-grams.

Overview of approach: q-grams are essential in understanding our technique to support pattern queries. q is a variable parameter that the user can change, which determines the number of q-grams per string. CryptInd++ indexes such q-grams. CryptInd++ maintains index entries that contain information about keywords which have a q-gram in common. Given a pattern query P_q, CryptInd++ first calculates all the q-grams in P_q. For all such q-grams, CryptInd++ fetches the keywords that contain at least one of the required q-grams. The keywords fetched are then checked locally at the trusted client side if they match the pattern P_q. Let K_p be the set of keywords that match the pattern. For all the keywords in K_p, similar to the technique illustrated in CryptInd, the document ids that contain the keywords are retrieved from the server and subsequently the required documents. We will now explain all the above steps in greater detail.

BuildIndex(CryptInd++): CryptInd++ contains two types of index entries: a) Keyword index entries; b) q-gram index entries.

Keyword index entries: Keyword index(KI) entries are similar to the CryptInd index entries. KI entries are triples $< id, H_i, B_i, E_{MP}(KW_i) >$. H_i and B_i have the same value/semantics as their namesakes in CryptInd entries. id refers to keyword entry id that is assigned to every keyword entry. At the beginning of BuildIndex, a counter is set to zero. For every keyword being indexed, the counter is incremented and its value is stored as the id for the keyword entry. $E_{MP}(KW_i)$ represents the actual keyword encrypted using masterpassword as the key. We will refer to it as the *encrypted keyword* part of the keyword index entry. Similar to cryptInd, for every unique keyword, a keyword entry is created.

q-gram index entries: A q-gram entry is a tuple $< H_i^q, B_i^q >$. H_i^q represents a keyed hash output of a q-gram q_i that is a substring to at least one keyword stored in the file system. B_i^q is a set of pointers which point to a keyword index entry. B_i^q contains the ids of the keywords which contain the substring q_i. We will refer to B_i^q of a q-gram entry as its *pointer set*. Care is taken so that the pointer sets of all the q-grams are of equal cardinality. A procedure similar to the one that maintains equal size of bitmaps is also followed here. As we will show later, this is done to ensure security. B_i^q is encrypted before being stored at the server.

Storing the index: Similar to CryptInd, both the keyword and q-gram index entries are mapped to objects and then stored at the server.

Search: Let $Q^p = \{Q_1^p, Q_1^p, Q_n^p\}$ be a query which a set of patterns. The objective is find all the documents that contain the words that match all the patterns in Q^p.

For every pattern $Q_i^p \in Q^p$, let Q_{iq}^p represent the set of q-grams in the Q_i^p. For every q-gram in Q_{iq}^p its hash $h_k(Q_{iq}^p)$ is calculated using the masterpassword as the key and corresponding q-gram index entry is retrieved from the server. Let the set $IQ = \{IQ_1, IQ_2, \ldots IQ_m\}$ represent all the index entries retrieved from the server in response to the query Q^p. All the bitmaps of the index entries in IQ are decrypted. Let $I.B_i$ represent the *pointer set* of q-gram index entry I. Then, a union of all the Bitmaps in IQ, i.e. $\cup_{j=1}^{m} IQ_j.B_i$, is calculated. $\cup_{j=1}^{m} IQ_j.B_i$ contains information about the keywords that could potentially satisfy the query Q^p. For every pointer in $\cup_{j=1}^{m} IQ_j.B_i$, the corresponding keyword entry is retrieved from the server. Let KW_I represent the set of keyword entries retrieved from the server. Now DataGuard checks for the keywords in KW_I that match atleast one of the patterns in Q^p. This is achieved by decrypting the encrypted keyword part of the index entries in KW_I. For all the keywords in KW_I that match, a procedure similar to the search in cryptInd is followed to retrieve the metadata of the files that contain the required patterns. The user can now choose to download any files that he/she desires.

Updates: The procedure to updates does not change much with respect to cryptInd. If a new keyword index entry needs to be added, then new q-grams entries also might need to be added, since the required q-grams may not be present in the current q-gram index entry set.

6.2 The value of q

For a string S of length, there are $(n - q + 1)$ q-grams. The proof is by mathematical induction. In the worst case, the number of q-grams will far outnumber the keywords.

In CryptInd++, we only index unique q-grams. In practice, they tend to be less than the keywords indexed (for q =3). Increasing the value of q potentially decreases the number of q-grams that need to be indexed. On the other hand, it reduces the flexibility in generating the pattern queries to the user. In DataGuard, the value of q is set to 3.

6.3 Analysis

6.3.1 Security Analysis

Claim 6.2 *CryptInd++ is semantically secure.*

Proof Sketch: *CryptInd++ contains two different types of entries. We now need to prove that both types of entries are semantically secure. From claim 6.1 it should be clear that keyword entries are semantically secure. The q-grams entries are also semantically secure, since given two q-gram entries, the attacker cannot difference between the two. Like the keyword entries, q-grams also employ a hash function and an encryption function internally, which are semantically secure.*

Song et.al. [7] describe three essential properties for a cryptographic search technique. We will now informally show that CryptInd++ satisfies all the three properties.

Hidden Queries: This property states that server should not know the keyword being queried. In our scheme, the actual keyword is never revealed, only its hash value is sent to the server.

Controlled Searching: This property states that the server should not be able to generate trapdoors for any given keyword. In our scheme, the hash value of the keyword that is sent to the server is the trapdoor. Since we use a masterpassword during the hashing process, the server cannot compute trapdoors locally.

Query Isolation: This property states that the server should learn nothing about the documents other than the search results. This follows directly from the CryptInd++ semantic security result that we proved previously. The index does not reveal any information at the server side.

6.3.2 False positive analysis

In our search technique, to answer a keyword query, some extra information is fetched from the server. This section quantifies the extra information/false positives fetched. Note that such false positive information is filtered at the client side. In CryptInd++, given a pattern query Q, we first extract all the q-grams in Q. Let q_s represent the set of such q-grams. Then, q-gram entries that represent the q-grams in q_s are fetched. Let q_s^I represent such q-gram entries.

Then, an intersection of all the keywords that are indexed in q_s is calculated and there corresponding keyword entries are fetched from the server. Let K_S represent the set of keyword entries that are fetched from the server. Now, there are three kinds of false positives that are possible in CryptInd++: a) The false positive pointers/references to keywords in q_s^I; b) Keywords in K_S that do not match the pattern/s in Q even though they contain all the q-grams in q_s; c) Keywords in K_S that do not contain all the q-gram in q_s.

Case a: A q-gram entry in q_s^I could contain references/pointers to keywords that do not satisfy all the q-grams in q_s. We will now try and quantify suck keyword pointers. Let $|q_s^I| = k$. Then, the number of false positive pointers are equal to:

$$KW(q_1) + KW(q_2) + \ldots KW(q_k) - KW(q_1 \cap q_2 \cap \ldots q_k)$$

Where function $KW(q_i)$ represents the cardinality of the pointer set belonging to the q-gram index entry that represents the q-gram q_i. Similarly, $KW(q_1 \cap q_2 \cap \ldots q_k)$ represents the keywords references/pointers that are found in all the q-gram entries in q_s. In this case, there is a potential for a large number of false positives. The false positives monotonically increase with the increase in q-grams in the pattern query.

Case b: It is very difficult to quantity the number of keywords in K_S that do not match the pattern/s in Q, since it largely depends on the dataset. We will first quantity the expected number of keywords in a dataset that contains all the q-grams in q_s. Then, we will show that such a number decreases exponentially with increase in q-grams. This result provides the intuition that while it is possible that a significant number of false positives are fetched from the server when the number of q-grams in q_s are low, in a fairly typical case the false positives are not that significant.

Let $|q_s| = n$. Also, let P_j be the the probability that a g-gram q_j appears in a keyword w.r.t to the dataset that is being indexed. Then, the probability that all the q-grams in q_s appear in a keyword is $\prod_{i=1}^{n} P_i$. Let N_k be the number of the unique keywords in the dataset. Therefore, the expected number of keywords that contain all the q-grams in q_s is equal to $N_k \times \prod_{i=1}^{n} P_i$.

In the worst case, all the $N_k \times \prod_{i=1}^{n} P_i$ keywords can be considered as false positives, i.e, they do not satisfy the pattern in Q. Since probability that a q-gram is present in a keyword is inversely proportional to the number of the keywords, the quantity $N_k \times \prod_{i=1}^{n} P_i$ decreases exponentially with increase in number of q-grams present in q_s. Therefore, false positives decrease exponentially with the increase in the q-grams in the pattern Q.

Case c: Keywords in K_S do not necessarily have to contain all the q-grams in q_s. Since hash collisions are possible, two or more q-grams can map to the same hash value. Therefore, keywords in K_S can contain other q-grams which map to the same hash value to that of q-grams

in q_s.

We will now analyze the probability that there exists a keyword in K_s that does not contain all the q-grams in q_s. Let N_q be the number unique q-grams in file system/dataset. Also, let S_h be the size of the hash function output in bits. Then,

$$Pr[\text{a given q\_gram q collides}]=\frac{(N_q-1)}{2^{S_h}}$$

$$Pr[\text{q-gram q does not collide}]=1-\frac{N_q-1}{2^{S_h}}$$

$$Pr[\text{ No collision for all the q-grams in } q_s]=(1-\frac{N_q-1}{2^{S_h}})^{|q_s|}$$

$$Pr[\text{false positive}]=1-Pr[\text{ No collision for all the q-grams in } q_s]$$

$$=1-(1-\frac{N_q-1}{2^{S_h}})^{|q_s|}$$

$$=1-e^{\frac{(N_q-1)|q_s|}{2^{S_h}}}$$

In DataGuard we use the SHA512 cryptographic hash algorithm which outputs a 512 bit long string. Hence $S_h = 512$. For any reasonable value of N_q and q_s, the probability of a false positive is nearly equal to zero.

In summary, CryptInd++ is a semantically secure cryptographic index that allows the DataGuard middleware to search for keywords on encrypted data. The novelty of CryptInd++ lies in the fact that no changes are necessary at the server side and it supports pattern queries, unlike the previous approaches.

7 Experiments

The goal of our experiments was to measure the performance of our system. DataGaurd's scalability is not much of an issue, since scalability depends on the systems maintained by the Internet storage providers. More concretely, we wanted to measure the performance of our middleware in enforcing data confidentiality constraints by encrypting/decrypting data objects, calculating data integrity information, and the network costs associated to it's usage.

Experimental Setup: Our experiments were conducted on the three layers that we mentioned before. The DB layer utilizes a IBM DB2 database as the storage server. This database was running on an 8 processor pentium machine with 32 GB Ram in our laboratory. The Gmail layer was written for a major commercially available web based email provider (Gmail web service). For the experimental data, we used a local file system of one of the authors. The file system was outsourced via the DataGuard middleware to both the storage servers. We accessed the data via a laptop machine containing an Intel Celeron(R) 1.80 Ghz processor with 786 MB of RAM.

Results: Our first experiments were to measure the network costs. Figures 4 and 5 report the file transfer/download times for all the three layers when the files were accessed from a laptop.

Not surprisingly, transfer times increase linearly with the file size. Since the DB layer was using a database that was geographically very close to the laptop, we are getting excellent file transfer rates. The transfer times for the Gmail layer were slower than that of the Amazon layer. One interesting thing to note was, for the Gmail layer, the download times were slightly slower than the upload times. The reason is the following: For storing/fetching files, the translational layer needs to send HTTP requests to the web based email provider. While, one HTTP request is sufficient for storing a file, it required 3 HTTP requests for fetching a file. First we had to use the search interface given by the email service provider to locate the email with the required object and then fetch it.

Our second experiment was to measure the cryptographic costs of DataGuard. Figure 6 reports the different cryptographic costs in DataGuard . The encryption times were very similar to the decryption times and they exhibit a very linear behavior. The integrity costs were lower than the encryption costs. The interesting things about this experiment was that fact the when compared to the network costs, the *cryptographic costs* pale in comparison. Therefore, the security itself is not degrading the performance of DataGuard by much. To confirm this intuition we conducted another experiment to measure the overhead for using DataGuard. We used cockpit, a tool that allows users to upload files to their Amazon S3 account. We measured the time taken to perform some standard file system operations via cockpit and DataGuard. Fig 7 reports the relative times. Not surprisingly, with DataGuard takes longer than cockpit to execute the operations, but the difference is negligible. This implies there is *no major performance degradation due to DataGuard*.

Our final experiment was to measure the performance of *CryptInd++* cryptographic index. We ran different pattern search queries and measured the time taken to answer the queries across all the three layers. Fig 8 reports the relative index search times for the three layers. Index search is very fast in DataGuard. The DB2 layer has the best performance, due to its close proximity to the client machine. In the Amazon layer, most search queries can be answered close 2 secs. The Gmail layer is the slowest of all the three layers. The reason is the following: The Gmail service temporarily disbands an account if a lot of emails are sent to the server with a small time period. Due to this reason, we were forced to store a large number of index entries in one email. Ideally, we would have stored one index entry in an email. Due to this restrictions, a significant amount of the CryptInd index needs to be fetched from the server to answer the query. Hence, the degradation in performance when it comes to Gmail. There are no such

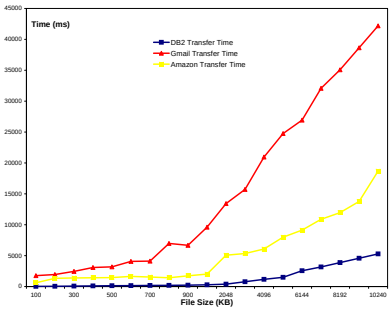

Figure 4. Network costs, transferring files

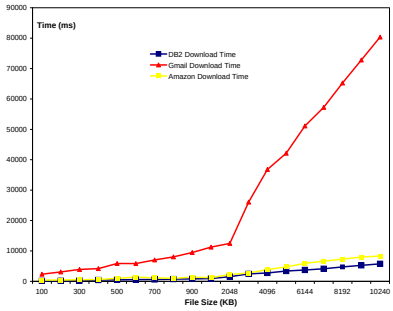

Figure 5. Network costs, downloading files

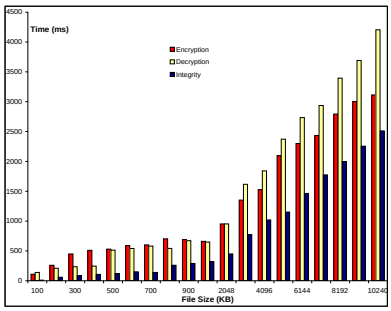

Figure 6. Cryptographic costs

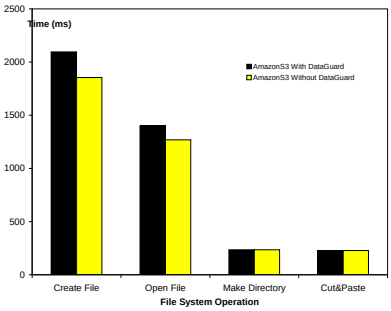

Figure 7. Overhead of DataGuard

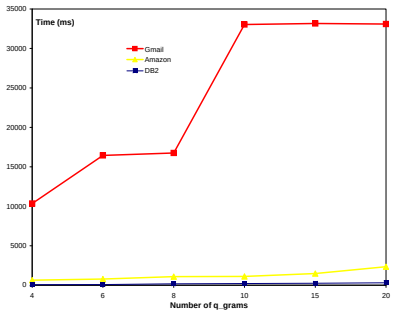

Figure 8. CryptInd++ Search

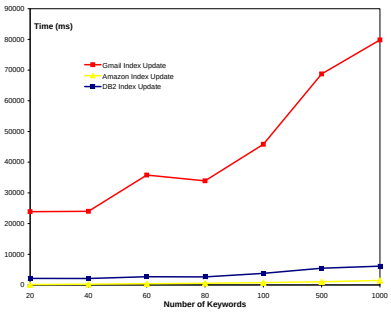

Figure 9. CryptInd++ Updates

issues with Amazon service and hence there is a noticeable improvements in the performance.

In the next experiment, we wanted to measure the cost of updating the index. A similar trend also exists in Index Updates. Fig 9 shows the results. The Gmail layer is much slower than the Amazon layer. To update 1000 keyword entries, it takes 1.4 secs in Amazon and 79 secs in Gmail. Since with some IDPs, the updates could be slow, there is a requirement to explore smart update algorithms that bulk update the index. This way, the client does not have to wait for a significant amount of time every time he updates a file. Although, this process generally can be in background, when the client machine is idle.

Another interesting experiment was to measure the q-grams/keywords ratio. Fig 10 illustrates the results. The number of q-grams indexed decrease as the keywords indexed increased. This is due to the fact that as keywords increase they tend have more and more q-grams in common.

Under Construction

8 Related and Future work

8.1 Related Work

Network file systems [18, 19, 20] allow users to outsource their information to a remote server. An authorized client can then mount the file system stored at the server. Typically in these systems, the server is trusted and is in charge of authentication of the users and enforcing access control on data. This is not the case in DataGuard.

Cryptographic file systems [2, 15, 13, 14] on the other hand are very related to our work. Cryptographic file systems do not trust the end storage and all the cryptographic operations are done at the trusted/client side. Cryptographic file systems such as Sirius [13] and Plutus [14] also allow sharing of files between users, where access to files is provided via key distribution. DataGuard currently does not deal with sharing, although it is one of our future goals. We differ from the cryptographic file systems in the following manner: a) cryptographic file systems do not adopt to the heterogeneity of data models of the server side. Typically, they assume a file system based model at the server. DataGuard on the other hand can easily adapt to the heterogenous data models at the server.

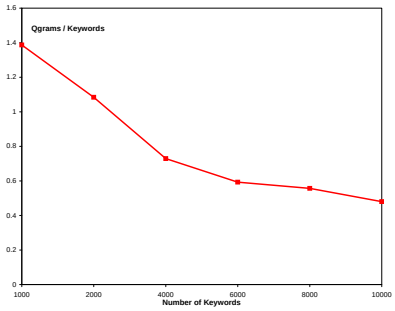

Figure 10. Q-gram/Keyword ratio

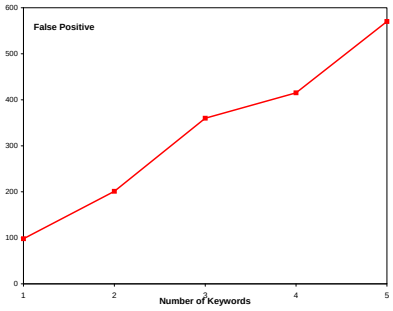

Figure 11. False positives in q-gram index entries

DAS [11, 12] architectures allow clients to outsource structured databases to a service provider. The service provider now provides data management tasks to the client. The work on DAS architecures mainly concentrated on executing SQL queries over encrypted data. The clients of DAS architectures are mainly organizations that require database support. Both the DAS architectures and DataGuard can be thought of as instantiations of the outsourced database model (ODB). The key differences are: a) The data outsourced in DAS is highly structured. In DataGuard, the data oursourced is semi-structured. b) DAS architectures did not deal with mobility issues, which is one of the primary goals of DataGuard.

Distributed file systems like oceanstore [10] provide a storage infrastructure for the users to store data on the network rather than at a centralized server. In ocenstore, the files are treated as objects and are replicated across multiple locations. The goal is to ensure availability, scalability and fault tolerance. DataGuard should not be treated as a distributed file system, since middleware treats the storage providers as a single logical entity. This does not imply that the service providers do not implement a distributed storage infrastructure. On the other hand, DataGuard does allow users to mount different file systems with multiple storage providers.

In DataVault [21], the authors proposed a client-server architecture which allows users to outsource their file systems to an untrusted server. The server then provides data services on top of outsourced data. DataGuard is not a client-server architecture, it is a middleware that is trying to utilize the storage space provided by untrusted servers on the Internet. In DataVault, the authors were able to design a server architecture from scratch that suits their data storage requirements. DataGuard middleware on the other hand has to work/adapt to the current data storage infrastructures of the IDPs.

Jungle disk software[**?**] layers a security mechanism over the Amazon S3 storage service. Unlike DataGuard, Jungle Disk can only function with the Amazon S3 service. Jungle Disk also provides a file system like interface to the user and preserves data confidentiality of the user by encrypting the data stored remotely. The user can provide a password as the key to encrypt the data. To the best of our knowledge, Jungle Disk does not verify the integrity of the data.

8.2 Future Work/Open Problems

In this section we will describe some of the open problems in DataGuard. The following problems will be the main focus for our future work in DataGuard.

Accessing Information from Untrusted Machines: Recall that we have initially made an assumption that all end devices the user access his/her data are trusted. While in most cases this is true, in some cases it isn't. For instance, consider Alice who is traveling without a laptop. She needs to access her data from a publicly accessible machines such the ones that are avaialble in cybercafe or a public machine. Such public machines can harbor macicious entities which could steal Alice's masterpassword. A simple keystroke logger will accomplish the job. Clearly, this is undesirable. We need techniques to to access personal data from untrusted data. In [1] the authors propose a proxy based solution to access websites securely from untrusted machines. We envision a similar solution could solve the above problems.

Sharing of Documents: In this paper, we only considered the problem of accessing data remotely. When it comes to personal information another requirement is the ability to share it. Currently, users use a variety of ways to share data: a) via email; b) running public servers, etc. DataGuard middleware could potentially could be extended to allow data sharing as well. This raises many challenges interesting challenges, since to enable data sharing the untrusted servers should authenticate users and distribute data. This needs to be done in a fashion where the server does not learn any user's data.

Building applications on DataGuard's framework: In the current avatar, DataGuard allows users to outsource their file system. There are many data services that can be built on top of such framework. For instance, consider autofill information of browsers. Such information is typically maintained as a file in the local hard drive. If the user allows DataGuard to outsource such files, then DataGuard can fetch the autofill information and install it at the appropriate place without bothering the user. Thereby, the user can now have his passwords, usernames, etc. automatically filled out wherever he/she goes. We will exploring such applications in the context of DataGuard.

9 Conclusions

In this paper we presented DataGuard, a middleware that allows users seamless access to their data from heterogenous data storage providers on the Internet. DataGuard ensures the confidentiality and integrity of the user's data. DataGuard utilizes a novel index based approach to allow keyword based search on encrypted data. Our main goal is to release the DataGuard middleware as an open source software that will allow experts or webmaster to write translation layers compatible with DataGuard. This will allow storage providers to provide data services on top on DataGuard. We have currently developed a prototype, which we intent to thoroughly test before releasing it to the public. A beta version of the DataGuard software can be downloaded at http://DataGaurd.ics.uci.edu.

References

[1] Ravi Chandra Jammalamadaka; Timothy van der Horst; Sharad Mehrotra; Kent Seamons; Nalini Venkatasuramanian. Delegate: A Proxy Based Architecture for Secure Website Access from an Untrusted Machine. 22nd Annual Computer Security Applications Conference (ACSAC), Maimi, FL, December, 2006

[2] M.Blaze. A cryptographic file system for UNIX. Proceedings of the 1st ACM conference on Computer and communications security.

[3] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava. D.Thomas, Y.Xu. Two Can Keep a Secret: A Distributed Architecture for Secure Database Services.2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005.

[4] Eu jin Goh. Secure Indexes. In submission

[5] RSA Laboraties. PKCS #5 V2.1: Password Based Cryptography Standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2_1.pdf

[6] Ravi Chandra Jammalamadaka, Roberto Gamboni, Sharad Mehrotra, Kent Seamons, Nalini Venkatasubramanian. DataGuard: A Middleware Layer Providing Seamless Mobile Access to Personal Data via Untrusted Servers. Techincal Report

[7] D. Song, D.Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In 2000 IEEE Symposium on Research in Security and Privacy.

[8] Briney, Andrew. 2002. The 2001 Information Security Industry Survey 2001 [cited October 20 2002]. http://www.infosecuritymag.com/archives2001.shtml

[9] Dhillon, Gurpreet, and Steve Moores. 2001. Computer crimes: theorizing about the enemy within. Computers & Security 20 (8):715-723.

[10] S.Rhea, P.Easton, D.Geels, H.Weatherspoon., B.Zhao, and J.Kubiatowicz. Pond: The oceanstore prototype. In the proceedings of the Usenix File and Storage Technologies Conference(FAST) 2003.

[11] Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. *2002 ACM SIGMOD Conference on Management of Data, Jun, 2002.*

[12] E.Damiani, S. De Capitani Vimercati, S.Jajodia, S. Paraboschi, P.Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. Proceedings of the 10th ACM conference on Computer and communications security.

[13] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003.

[14] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in Proc. 2nd USENIX Conference on File and Storage Technologies (FAST), 2003.

[15] E.Zadok, I.Badulescu, and A.Shender. Cryptfs: A Stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.

[16] G.Miklau, D.Suciu, Controlling Access to Published Data Using Cryptography. VLDB 2003: 898-909

[17] E.Bertino, B.Carminati, E.Ferrari, B.Thuraisingham and A.Gupta. Selective and authentic third party distribution of XML documents.

[18] S.Shepler, B.Callaghan, D.Robinson, R.Thurlow, C.Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3530, April 2003.

[19] J.Howard. An overview of the andrew file system. In proceedings of ACM symposium on parallel algorithms and architectures. SPAA, 2002.

[20] David Mazires. Self-certifying file system. Phd Thesis. 2000

[21] R.Jammalamdaka,S.Mehrotra, K.Seamons, N.Venkatasubramanian. Providing Data Sharing as a Service. Technical Report.

[22] http://www.aws.amazon.com/s3

[23] http://www.apple.com/dotmac/

[24] Ravi Chandra Jammalamadaka, Roberto Gamboni, Sharad Mehrotra, Kent Seamons, Nalini Venkatasubramanian. gVault. A Gmail Based Cryptographic Network File System. To appear in the proceedings of 21st Annual IFIP WG 11.3 Working Conference on Data and Applications.

10 Appendix

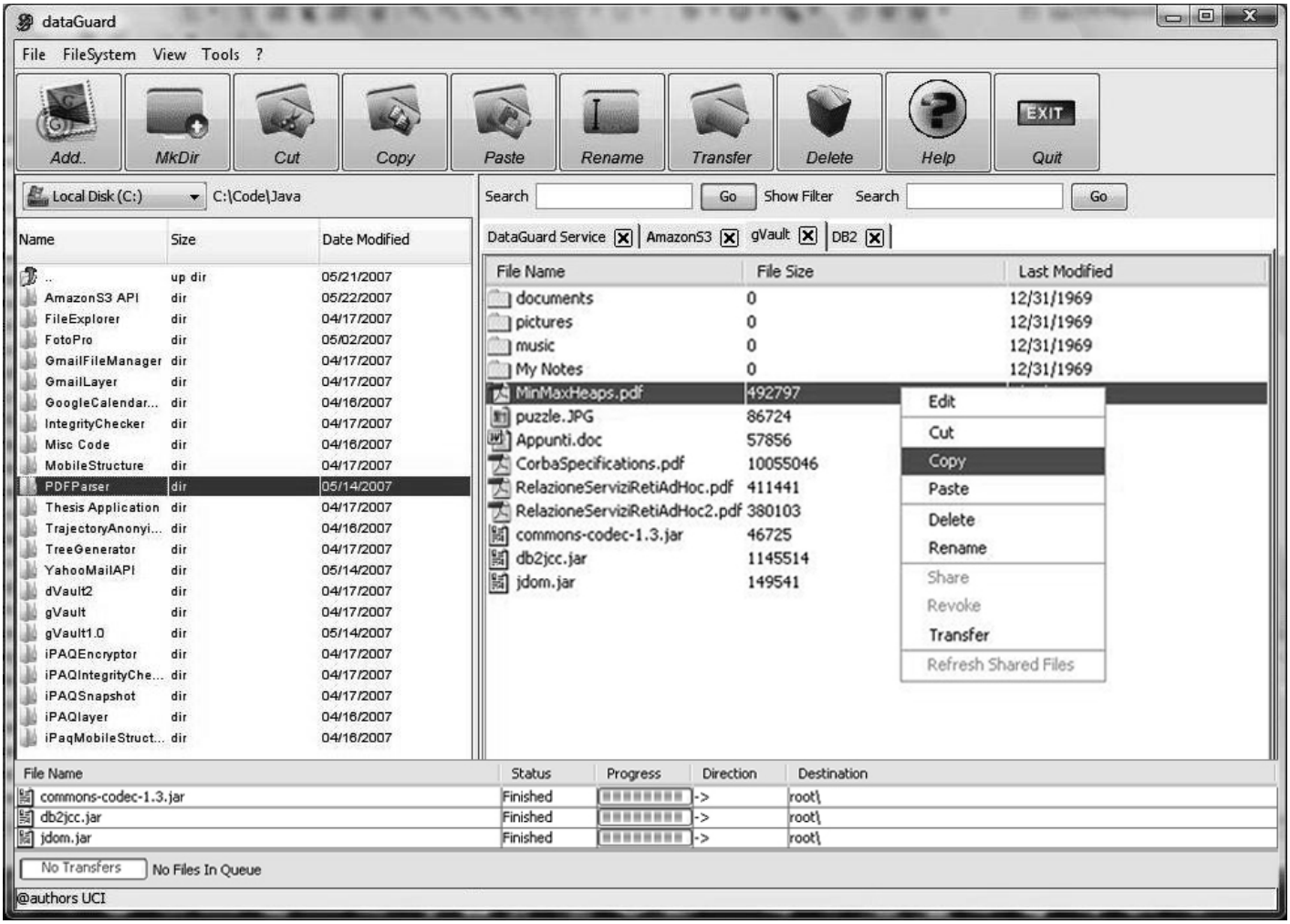

Figure 12. Snapshot of the DataGuard application/middleware. The interface is similar to the one provided by the modern operating systems. DataGuard is implemented in Java.