gVault: A Gmail Based Cryptographic Network File System

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 gVault, a cryptographic network file system that utilizes the data storage provided by Gmail's web-based email service. Such a file system effectively provides users with an easily accessible free network drive on the Internet. gVault provides numerous benefits to the users, including: a) Secure remote access: Users can access their data securely from any machine connected to the Internet; b) Availability: The data is available 24/7; and c) Storage capacity: Gmail provides a large amount of storage space to each user. In this paper, we address the challenges in design and implementation of gVault. gVault is fundamentally designed keeping an average user in mind. We introduce a novel encrypted storage model and key management techniques that ensure data confidentiality and integrity. An initial prototype of gVault is implemented to evaluate the feasibility of such a system. Our experiments indicate that the additional cost of security is negligible in comparison to the cost of data transfer.

1 Introduction

Network file systems have become quite popular in the past two decades. In such systems, user data in the form of files is stored at a remote server. The server is then in charge of providing services such as backup, recovery, storage, access, etc, thereby absolving the user from it's responsibility. The user can then *mount* the remote file system as a *local drive* and proceed to perform all the required file operations on the remote data. The biggest advantages of network file systems is that they allow users to remote access their data. A nomadic user can connect to the remote server from any machine connected to the Internet and access his information.

A related trend is the rise in popularity of web based email service providers (WESPs). Such services provide the users with the facility to send/receive emails free of charge. The business model is typically based on advertisements that are displayed on webpages the user is currently accessing. A big advantage of such systems, like network file systems, is that they allow the user to access his email from any machine connected to the Internet. Typically, WESPs allocate a lot of

storage to the user to store his/her emails, which emphasizes the fact that online storage has become very cheap.

Imagine a network file system built over the storage offered by the WESPs. Such a file system has numerous advantages which include: a) Remote access: The users can access their data from any machine connected to the Internet; b) Availability: The data is available 24/7; and c) Storage capacity: The WESPs provide a large amount of storage space to the user. Such file systems already exist. GMail Drive [1] and RoamDrive [2] are examples of applications that layer a file system over the storage space provided by Gmail, a prominent WESP on the Internet. Google's usage policy [4] does not prevent such file systems to utilize their system for data storage.

The biggest drawback of the current systems is the inherent *trust* they place on the WESPs. The data is stored in plain text at the WESP and is vulnerable for the following attacks:

- **Outsider attacks:** There is always a possibility of Internet thieves/hackers breaking into the WESP's system and stealing or corrupting the user's data.
- **Insider attacks:** Malicious employees of the WESPs 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 the attacks are insider attacks [11].

There also have been instances of WESPs collaborating with repressive regimes and providing them personal data that belongs to users [5]. Fear of prosecution is in itself a valid reason not to trust WESP with our personal data. As such, use of WESPs for storing potentially sensitive information has been heavily limited.

In this paper, we address the problem of designing a cryptographic network file system called gVault, that utilizes the storage space provided by the WESP and addresses the drawbacks of the earlier systems that we previously mentioned. gVault runs on top of the Gmail's email service and provides the users a file system like interface. gVault is fundamentally designed to be used by an average computer user and does not trust the Gmail storage servers with his/her data. The reader should note that while we have designed and implemented a system over Gmail, the techniques that we develop in this paper are applicable to any Internet based storage provider. We choose Gmail since it is widely popular and free to use.

There are many challenges that need to be addressed in designing a system of this kind. The first set of challenges occur due to the requirement of *designing a file system that is easy to use*. Therefore, we made a fundamental decision to control the overall security that is offered by gVault by a master password, a secret known only to the user. Passwords can be forgotten or stolen, therefore we need techniques that can defend against such situations. The second set of challenges occur due to users requiring their data remain confidential. In order to ensure that no unauthorized recipients obtain a user's data, data needs to be encrypted. This further raises many questions: a) What is the granularity of encryption? b) How is key management done? c) What is the encrypted storage model at the Gmail side? Such a model should optimize the data storage and

fetching operations at the server. The third set of challenges occur due to the *data integrity* requirements at the client side. Techniques/mechanisms should be in place that detect any data tampering attempts at the server side. The fourth set of challenges are *implementation* based. All the client side file operations need to be translated into HTTP requests that Gmail servers can understand.

Overview of our approach: gVault runs as an application on the user's machine. gVault prompts the user to enter his Gmail username, Gmail password and the master password. Once the users enters the information, gVault opens a session with the Gmail server and functions as an HTTP client. All the file operations that the user performs at the client side is mapped to their equivalent HTTP requests. The responses from the Gmail servers, which are encapsulated in HTML, are parsed to recover the required user's data. gVault implements the necessary cryptographic techniques to ensure the security of user data in the translation of file operations to HTTP requests. In our model, outside the security perimeter of client machine everything else is untrusted. We do not place any restrictions on the kind of attacks that can be launched.

Paper Organization: The rest of the paper is organized as follows: In section 2, we describe our encrypted storage model that provides data confidentiality. In section 2.1, we describe the relevant operations in the encrypted storage model. In section 3, we describe our techniques that allow the user to detect any tampering attempts at the server side. In section 4, we describe the implementation details of gVault. In section 5 and 6, we present the related work and conclude with future directions.

2 Encrypted Storage Model (ESM)

This section describes how a user's file system is represented at the Gmail servers. Our objective is to design and implement an encrypted storage model that preserves the *confidentiality* of user's data. More concretely, we want to design an encrypted storage model that does not reveal: a) *Content of the files*; b) *Metadata of the file system*; and c) *Structure of the file system* to the server. It is obvious that the file content must be protected. We believe that mere encryption of the files is insufficient in itself. Both the metadata and the structure of the file system also contain a lot of information about user's data and therefore it makes sense to hide them as well. Metadata of the file system contains information such as file names, directory names, etc., and file system's structure reveals information about the number of directories, the number of files underneath a directory, etc. If the storage model at the server side reveals the file structure an adversary can launch known plaintext attacks and discover information about the user's data.

Current network file systems models do not satisfy our requirements as they tend to reveal the structure of the file system to the server. For instance, the NFS storage model [17] maps every file and directory at the client side to a file at the server side, thereby revealing the structure of the file system.

The following defines our encrypted storage model:

Input: User's File System F
BEGIN
1. Decouple F into a File Structure
F_S and $S_F = \{ F_1, \ldots F_n \}$ a set of files.
2. Encrypt the File structure F_S and store it at the server.
3. For every file f_i in S_F
    Generate the object encryption key k_i
    Encrypt the file f_i with k_i
    and store it at the server.
END

Fig. 1. Encrypted storage Model

Fig. 2. Example of File Structure

Definition 1. ***File System:*** *A user's file system is represented as a graph $G =< V, E >$, where V is a set of nodes that represents both files and directories. The set E represents the set of edges between the nodes. Let the function $parent(n_1)$ represent the parent node of node n_1. If node n is the parent of node n_1, we present the relationship as follows: $n \leftarrow parent(n_1)$. The edge set E contains all the edges between any two nodes n_1 and n_2, where $n_1 \leftarrow parent(n_2)$ or vice versa. For every node $n \in V$, $n.metadata$ represents the metadata that is associated with the node n and $n.content$ represents the content of the node.*

For a file node, its metadata is the name of the file, size of the file and last modified time. For a directory node, its metadata is the name of the directory, number of files and directories underneath it, size of the directory and last modified date. The content of the directory node is set to null.

We could have modeled the file system as a tree. Then such a model does not take into account the *symbolic links* that are present in the file system. The graph structure allows the incorporation of the symbolic links in the file system.

Definition 2. ***Server Side representation of the user's file system:*** *Let the user's file system F be a graph $G =< V, E >$. A user's file system F is represented as a tuple $< F_S, S_F >$ at the server , where F_S is the file structure of the file system and S_F is a set of file nodes $\{F_1, F_2, \ldots F_n\}$ that belong to the file system. The file structure F_S is a graph $G' =< V', E' >$, where V' contains all the nodes in V and E' contains all the edges in E. For all the file nodes $n \in V'$, $n.content$ is a pointer to the relevant file in S_F.*

Fig 2 shows an example that illustrates the content of a file structure. We have decoupled the file system into a file structure object and a set of files. We will store these two data components separately at the server. As the server is untrusted, both F_S and S_F are encrypted before being stored at the server side. Fig 1 describes our overall mapping strategy to transform the plaintext at the client side to the ciphertext at the server side. For the rest of the paper, we will refer to both the file structure and individual files as data objects.

2.1 Operations in the ESM

Key generation: In gVault, the key to encrypt/decrypt the data object is generated on the fly depending on the metadata that is attached to the object. Using the metadata, gVault generates an object encryption key (OEK) for every object that is outsourced to the server.

We use the key derivation function (KDF) of the password based encryption specification PKCS #5 [8] to generate OEKs. 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 gVault, the OEK K_{fs} for the file structure object is calculated as follows:

$$K_{fs} = KDF(MasterPassword, Salt, Iteration)$$

The OEK K_i for each file object is calculated as follows:

$$K_i = KDF(FileName||MasterPassword, Salt, Iteration)$$

The password paramater in the KDF function is the concatenation of the filename and the master password. Salt is a large random string that is generated the first time the object is created and is stored in plaintext along with the encrypted object. The filename does not include the full path name to permit a file objects to be moved between directories without changing its OEK. The iteration count is set to 10000, the recommended number.

The primary reason that we generate a key for every individual data object is to prevent cryptanalysis attacks, whose effectiveness increases with the amount of ciphertext available that is encrypted with the same key. 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 password. We chose to generate the key since the KDF function is inexpensive compared to retrieving a key along with each object from the server. This reduces network bandwidth requirements, especially for small objects when the cost of retrieving the key would dominate.

Updating the file structure: The biggest drawback with our solution of decoupling of the file structure is that updates are not easy to handle. If the user makes changes to the file structure, we need to update the file structure stored at the server side. Updating the complete file structure for every update

will reduce the performance of the system and make the user wait for a relatively long time.

gVault utilizes a novel approach similar in spirit to the approaches in journaling file systems to combat the above issues. gVault maintain a log of all the updates that take place in a session. Let L represent the log which is a set of log entries $\{L_1, \dots L_n\}$. Each log entry L_i represents an update operation on the file system's structure. Update operations include cut, copy, paste, create and rename. For brevity, we will not describe the language used for representing such update operations. Whenever an update takes place, an appropriate log entry is added to the log and the file structure is updated locally. Let us assume that a user by issuing updates represented by the log L, has transformed the initial file structure F to $F^{'}$. When the user decides to log off, gVault encrypts $F^{'}$ and updates the file structure stored at the server side. After it successfully updates the file structure, gVault removes the log. If the application running at the client crashes, due to power failure or a hardware failure, etc., the gVault upon restart will look at the log L and reconstruct the file structure $F^{'}$ from F the last committed copy.

Notice that we primarily maintain the log for updates on the file structure. For updates which involve file transfers, we do not actually store the file contents on the log. Rather, a message is inserted into the log stating that a file transfer operation on particular file has started. When the file transfer is finished, a log entry is added to state the same. If due to a system crash, a file transfer is stopped midway, gVault will look at the log and figure out the failed operation and alert the user. It is up to the user now to take appropriate action.

Master password management: It is of paramount importance that a user does not reveal his/her master password to anyone. The loss of a master password could lead to disastrous effects, since now the adversary can have complete control over the user's file system. Passwords are prone to be lost or stolen. Therefore, there is a requirement to build mechanisms that change and recover the master password.

Changing the master password: Changing the master password will have its effect on the generation of keys. Hence, changing the master password could potentially be a very expensive operation, since it will require all the files/file structure to be decrypted with the old keys and encrypted again with the new keys. Fortunately, we can do this expensive operation *lazily.* gVault keeps track of both the new master password and the old master password in a configuration file that is encrypted with a key that is generated by the new master password. gVault will continue to decrypt the files that are fetched from the server with the keys generated from the old password. When a update is made to the file, then gVault will encrypt the updated file with the key that is generated with the new password. GVault will then set a flag in file structure to indicate that in future, it needs to use the key generated with the new master password. After a while, all the files will be encrypted with keys from the new master password.

Lazily changing the master passwords is typically done to periodically update the master password, a recommended practice.

In a situation where the master password is compromised, the user could request the change of keys immediately. This is an expensive operation, the user has no choice but to wait till all the files are fetched, decrypted with the old key and encrypted with the new key. If the user decides to change the master password more than once, gVault will again force the change of all the keys.

Recovering the master password: We have designed a novel approach to recover the master password in case the user forgets it. Unlike traditional password recovery mechanisms, in our case the master password is also not known to the server. When the user first utilizes the gVault service, the application prompts the user to enter a set of question/answer pairs. Let Q_a represent such a set, where an element $Q_i \in Q_a$, is a tuple $< \mathcal{Q}, \mathcal{A} >$ where $\mathcal{Q}$ represents the question and $\mathcal{A}$ represents the answer. The user can enter any arbitrary number of questions and answers. In other words, the user can control the cardinality of set $|Q_a|$. In our implementation, gVault suggests a few questions, but it up to the user to select some of the questions or come up with his own. After the user selects the questions, he/she will then proceed to submit the relevant answers to the questions. The answers to these questions will be kept secret from the server, while the questions are stored in plaintext. A user needs to be careful in his/her choice of questions. The answers to these questions should typically lead to information that only the user knows about. An example of a suitable question is *"what is my social security number"*. The user is assumed never to forget the answers to these questions. We derive an encryption key called the *Recovery key* from the answers to these questions. The recovery key is derived as follows:

$$Recovery_key = KDF(||Q_a, Salt, Iteration)$$

where KDF is the key derivation function discussed earlier. $||Q_a$ represents the concatenation of all the answers in the set Q_a. *Salt* is a random string that is stored in plaintext and iteration number is set to 1000. Now using the recovery key we compute *recovery information* RI of the master password as follows:

$$RI = E^k_{Recovery_Key}(MP)$$

where MP refers to the master password and E^k represents the encryption function applied iteratively k times. Typically k is set to a value which increases the encryption time to the order of seconds. This is done to reduce the effectiveness of the brute force attacks. The adversary now also has to spend considerably more time per brute force attempt. RI is then stored at the server. When the user forgets his master password, gVault will fetch Q_A from the server and present the user with the questions. Note, we assume the user remembers his Gmail's username and password. After the user answers the questions, the *Recovery_key* is recalculated and master password is recovered as follows:

$$MP = D^k_{Recovery_Key}(RI)$$

where D^k is the decryption function applied iteratively k times. Note that if the user does not provide the right answers, the recovered master password will not be the same as the original. The user can manually verify if that is the case, and can repeat the process if necessary. This process has similarities to the authentication protocols used by current websites, where an answer to a secret questions allows the user to recover a forgotten password. Therefore, the users are already familiar with such recovery mechanisms.

The password recovery is a useful feature, but it has the potential to be very insecure. Since the questions are in plain text, anyone with access to the server can learn the questions and launch a dictionary attack. This feature is only as secure as someone is able to make their question set strong against dictionary attacks. gVault provides a set of reasonable questions, the user would do well to select enough questions from this set to thwart a dictionary attack.

2.2 Analysis

Security: An adversary at the server side, by looking at the ciphertext stored at the server can procure some information regarding the file system of the user. The information includes: a) *The number of files and their relative sizes:* The size of the ciphertext dictates the size of the plaintext files; b) *The size of the file structure:* File structure is the first data object that is downloaded by the user upon login. The size of the file structure linearly increases with the number of nodes inside it. Hence, the adversary can reasonably guess the number of directories and files the user's file system contains. But, the adversary cannot find any information regarding the general hierarchy of the file system.

Another alternative for the encrypted storage model is to create a data object that subsumes both the file structure and the files. Such an object can then be downloaded at the beginning of the session. This representation provides more security, since the adversary does not know if the user has large number of files, or a large number of internal nodes in the file structure. Downloading the entire file system at the time of login puts an enormous performance strain on the system thereby making the system inherently not usable. The current design of the encrypted storage model strikes an appropriate balance between performance and security .

Performance: The file structure is fetched at the beginning of every session so that gVault does not need to contact the server while the user is navigating the file system. Adopting a storage model similar to the NFS model would force gVault to retrieve internal nodes on demand by visiting the server every time the user descends or ascends a level in the file structure. For slow Internet connections, the system will be less usable due to network latency at each navigation step. An improvement in the NFS model would be to allow the client to prefetch all of the internal nodes. Also, when the entire file structure is retrieved, the cost of decrypting the nodes individually is higher compared to a single bulk decryption of the complete file structure, since most encryption algorithms have a constant startup time. Thus, gVault encrypts the entire file structure as a single object.

It is possible that the user could pay a huge startup cost for downloading the complete file structure. The reader should note that this is a one time cost and in practice we have noticed that even for huge file systems, the file structure can be loaded quickly since its size tends to be very small.

Search: Another side benefit of the model is that it allows gVault to answer certain search queries. Since the complete file structure is cached locally, a client's queries on file and directory names can be executed locally without contacting the server. In order to conduct searches over the file content at the untrusted server, gVault would need to support searching over encrypted data. This is part of our future work, and we will explore existing solutions [7, 9] and determine how they can be adapted to gVault.

3 Data Integrity

Another requirement of gVault is that data integrity be preserved. This section describes how gVault ensures the *Soundness* and *Completeness* of a user's data.

Soundness: To ensure soundness of a data object, gVault needs a mechanism to detect when tampering occurs. To achieve this, the HMAC[1] of an object is calculated and stored in the file structure along with the object/file node. 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 of an object O is as follows:

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

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:

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

Every time the object is updated, the version number is incremented and the HMAC recalculated. This is done at the client side and hence there is no loss of security. In our model, the user is roaming and we do not store any data locally. Therefore, the impetus is on the user to validate the version number. Doing so, for every object he/she accesses will obviously make the system unusable. Fortunately, we can store the version numbers of all the file and directory nodes as metadata within the nodes themselves. That is, the version numbers are stored in the file structure. When the user accesses a file, gVault computes the HMAC of the file by using the version stored in the file structure and verifies whether it matches the HMAC stored in the file structure. If it matches, then it confirms

[1] A keyed-hash message authentication code

that the user has the right version. While such an approach will work, there is still the problem of validating the version number of the file structure. In gVault, the last modified date is used as the version number for the file structure. The user is now entrusted with verifying the version number of the file structure. The user needs to do it only once at the beginning of every session and it is the only version number the user must validate. This is by no means a complete solution since it requires some manual validation and a human is known to be the weakest link in security architectures, but it builds some defenses in detecting if data tampering has taken place at the server.

Another method is to calculate the *global signature* of the complete file system using a Merkle tree approach [3] 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 WESPs. 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 allows the client application to detect data tampering attempts at the server, without any user involvement.

Completeness: In gVault completeness is trivially achieved, since a client always knows the correct number of objects that need to be returned by the server. For instance, if the user is accessing a file, the server is required to return only one object.

4 gVault Prototype and Evaluation

This section describes the implementation details of the current gVault prototype. gVault is implemented in Java mainly for software protability reasons. gVault executes totally on the client side. No server side modifications are required. The current implementation of gVault is an executable jar file and hence does not require any installation. In the future, we will release gVault as an applet so that it can run from a web browser. Fig 3 illustrates the overall software architecture.

Components: The clients interact with gVault through the File System Local and the File System Remote components. The File System Local component provides a GUI interface to the local file system where the application is running. The File System Remote component provides a GUI interface to the file system stored at the remote server. The interface (see fig 4) of both these components is similar to the interfaces that exist to any modern file system. The Object Model Translator maps the file system the user is outsourcing into data objects. The Cryptographic Module supports the object model translator in the cryptographic operations. The HTTP Handler translates file operations into HTTP operations that Gmail servers support.

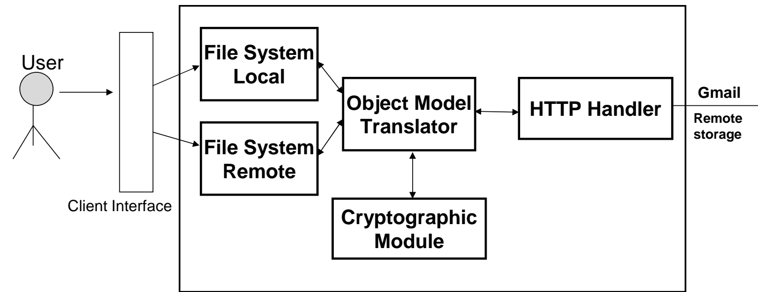

Fig. 3. gVault Software Architecture

User Interface and Functionality: Fig 4 illustrates the screenshot of the gVault system. Users of gVault must establish a username and password with the Gmail service prior to using gVault. During login, the user has to submit the username/password of Gmail and the gVault master password that is used to control the cryptographic operations. Obviously the master password must be strong, since the security that gVault provides depends on the master password. gVault allows the users to maintain multiple accounts thereby realizing multiple file systems. One of these accounts is designated as the *primary account* and this account stores the required information that will allow gVault to open all the file systems, when the user provides the credentials of the primary account. The different remote file systems are shown in different tabs and the user can switch from one file system to another with a simple click.

gVault supports prioritized execution of basic file operations for better interactive response times. The HTTP handler maintains two priority queues namely the *Operation Queue* and the *File Transfer Queue*. The HTTP handler utilizes these two queues to schedule the file operations issued by the user. The operation queue is primarily used to queue operations to which the user expects immediate response. Examples of operations include, deleting a file, opening a relatively small file, etc. The File transfer queue maintains operations which the user should expect a longer response time such as opening a large file. The HTTP handler opens two sessions with Gmail simultaneously to schedule the operations from each of the queues separately.

Another important feature of gVault is its ability to handle large files. Currently, the atomic unit of storage is a file. The Gmail service has a limit on the maximum size of the file that can be outsourced. When a local file exceeds the limit, gVault transparently breaks a large file into a set of objects that conform to the Gmail's limitations. When access to the file is required, gVault fetches all the required objects and combines them for the user.

gVault supports the standard set of file operations. A user can create/open files and directories, delete files/directories, etc. We will now explain some of the implementation details for these operations.

Creating a file: To create a file, gVault first adds a log entry that includes the name of the file. Then, gVault adds the file to the local file structure. The file

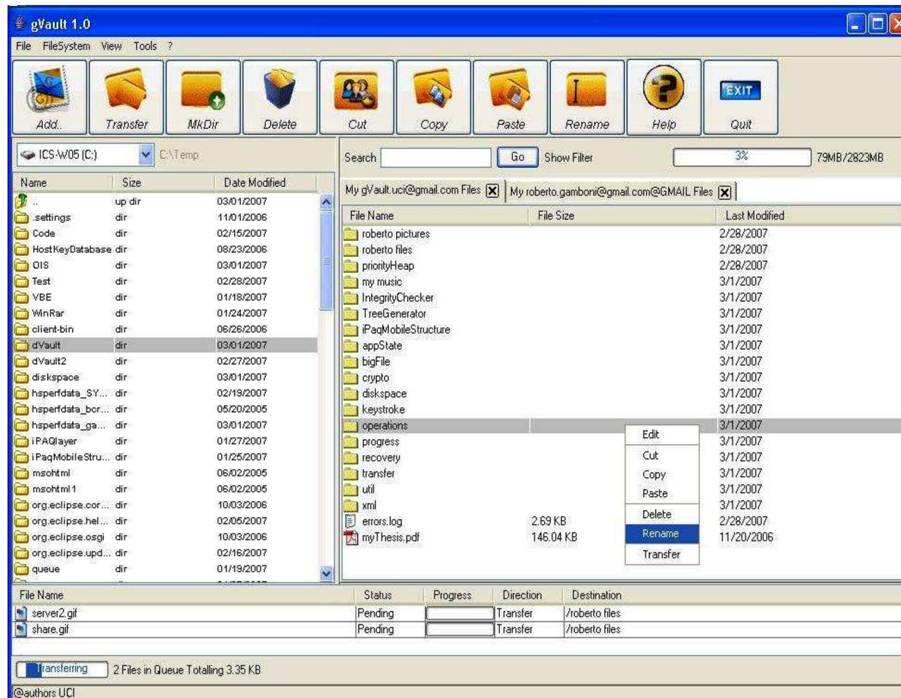

Fig. 4. gVault Screenshot

is then encrypted using its encryption key and stored at the server. To achieve this, gVault first calculates the file id by hashing the filename and the random salt generated during the key generation. Then, gVault creates an HTTP POST message that sends an email message to the user's Gmail account. The subject of this email contains the file id and the body of this email contains the encrypted file content.

Opening a file: To open a file, gVault must locate the corresponding email message containing the file. To accomplish this, gVault first calculates the file id. Then, gVault uses Gmail's search interface to retrieve the email according to the required file id in it's subject header. This requires that one HTTP POST request containing the search query (i.e., file id) be sent to the Gmail server. After gVault identifies the relevant email, it issues another HTTP GET message to retrieve the email. Once the email is fetched, gVault parses the body of the email, decrypts the file content and displays it to the user. Note that file creation requires one HTTP request, while opening a file requires two HTTP requests. Since gVault does not control how emails are stored at the Gmail server, it must search for an email message containing the desired file. .

Updating a file: To update a file, the client follows a similar pattern to creating a file. To create a file, the client needs to add a node to the file structure

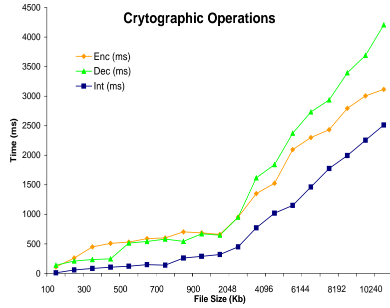

Fig. 5. Cryptographic operations

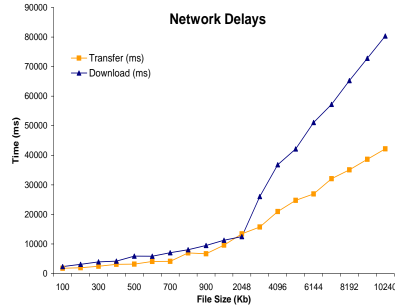

Fig. 6. Network delays in transferring files

locally. To update a file, the corresponding node already exists locally, so gVault encrypts the content and stores it on the server.

Moving a file: To move a file, gVault first adds the relevant update operation to the log at the server. In gVault, the combination of a file name and its random salt is unique. Therefore, the file encryption key does not change when an object moves. Using the pathname in the key generation process would force gVault to decrypt the object and encrypt it again with the new key. Under the current design, all gVault needs to do is to update the file structure.

Deleting a file: To delete a file, gVault first updates the log with the delete operation. Similar to opening of the file, gVault first identifies the email that contains the file by using Gmail's search facility, and then sends an HTTP POST message that deletes the email from the server. Then, gVault updates the file structure stored locally at the client.

Other operations: Additional operations such as moving a directory, renaming a file, deleting a directory, etc. are not described due to space limitations. The implementation of these operations follows a similar pattern to the operations described previously.

4.1 Performance

We conducted experiments to measure gVault's performance in encrypting/decrypting data objects, calculating data integrity information, prefetching the file structure, and the network delays for transferring files.

Our experiments were conducted on an Intel Celeron(R) 1.80 Ghz processor with 768 MB Ram client machine. For the experimental data, we used a local file system of one of the authors. This data contained a wide assortment of files such as video files, mp3 files, word documents, excel spread sheets, text files, etc. The file system was outsourced via the gVault application to Gmail storage servers.

Fig 5 describes the cryptographic costs associated with gVault's usage. The cryptographic costs includes the encryption costs, decryption costs, and the integrity costs. As expected, encryption and decryption costs are nearly the same

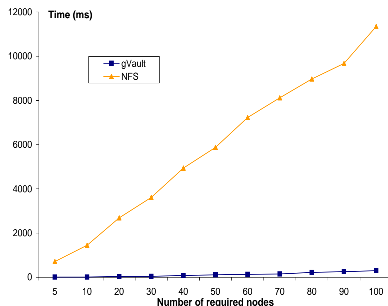

Fig. 7. Fetching the required nodes

and the cost of calculating integrity information is lower than that of the cryptographic costs. Fig 6 shows the network costs. Note that when compared to network costs, cryptographic costs are nearly negligible. For instance, to transfer a 10 MB file to the Gmail server, it takes about 85 secs. The cryptographic costs for the same file totally amount to 6.5 secs.

We wanted to measure the performance gain due to prefetching the complete file structure. If we were to follow an NFS based storage model, then every time the user descends a level in the file system, the server needs to be contacted to fetch all the child nodes. In Gmail, there is no API available to directly download the emails that contain the required nodes. Therefore, gVault uses Gmail's search interface to find the required emails and individually download them. Fig 7 shows the comparison of this approach to the NFS approach. gVault does significantly better since it can fetch all the required nodes locally. In summary, we conclude that enabling secure storage over web-based data storage providers is feasible and cost effective.

5 Related Work

Cryptographic file systems [14, 15, 6] provide file management to users when the underlying storage is untrusted. This is typically the case when data is stored at remote untrusted servers. Cryptographic file systems can be classified under two categories: a) password based; and b) non-password based. Sirius [14] and Plutus [15] are examples of cryptographic file systems that are non-password based. Their goal is to provide the user with data confidentiality and integrity when the data is stored at a remote server. We differ from them in the following manner: a) these systems were built for sophisticated users. For instance, in Sirius and Plutus, the users are expected to purchase a public/private key pair and securely transport it when mobile access is desired. Our architecture is catered to average computer users and we placed heavy emphasis on making the system easy to use without sacrificing security.; and b) the encryption storage model leaked the file structure information which is not the case in gVault.

There are other cryptographic file systems that are password based. Most notable of them are the cryptographic file system(CFS) for Unix [6] and Transparent cryptographic file system(TCFS) for Unix [18]. CFS is a user level file system that allows users to encrypt at the directory level and the user is supposed to remember a pass phrase for every directory he/she intends to protect. TCFS is in many respects similar to CFS, except for the fact that cryptographic functions are made transparent to the user. To the best of our knowledge, both these system do not provide mechanisms to recover passwords. Besides the recovery mechanisms, gVault also differs from these systems in the storage model.

DAS [12, 13] 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 architectures mainly concentrated on executing SQL queries over encrypted data. The clients of DAS architectures are mainly organizations that require database support. In this paper, our objective is to come up with a file system like service and hence we fundamentally differ from DAS related research works, even though we are similar in spirit.

6 Conclusions

This paper presented the design and implementation of gVault, a cryptographic network file system that provides a free network drive to the storage space offered by Gmail, a web-based email service. gVault protects the confidentiality and integrity of a user's data using cryptographic techniques. gVault provides users with a file system like interface and allows them to seamlessly access their data from any computer connected to the Internet.

gVault is designed for an average user. The overall security that gVault provides depends on the user remembering a master password. A mechanism is provided to change or recover the master password in case it is forgotten or stolen. This makes gVault usable for a wide spectrum of users. A beta version of gVault is available for download at http://gVault.ics.uci.edu.

7 Acknowledgements

We would like to acknowledge the work done by James Chou, Andrew Grant and Mark Lem who implemented parts of the gVault application. This research was supported by funding from the National Science Foundation under grant no. CCR-0325951 and IIS-0220069, the prime cooperative agreement no. IIS-0331707, and The Regents of the University of California.

References

1. Gmail Drive. *http://www.viksoe.dk/code/gmail.htm*
2. *http://www.roamdrive.com*
3. R. Merkle. Protocols for public key cryptosystems. IEEE security and privacy, 2000.

4. Gmail program policies. *http : //mail.google.com/mail/help/intl/en/program_policies.html*
5. Man Jailed after Yahoo Handed Draft Email to China. http://www.ctv.ca/servlet/ArticleNews/story/CTVNews/20060419/yahoo_jail_ap_060419/20060419?hub=World
6. M.Blaze. A cryptographic file system for UNIX. Proceedings of the 1st ACM conference on Computer and communications security.
7. E.j.Goh. Secure Indexes. In submission
8. RSA Laboraties. PKCS #5 V2.1: Password Based Cryptography Standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2_1.pdf
9. D.Song, D.Wagner, and A.Perrig. Practical Techniques for Searches on Encrypted Data. In 2000 IEEE Symposium on Research in Security and Privacy.
10. A.Britney. The 2001 Information Security Industry Survey 2001 [cited October 20 2002]. http://www.infosecuritymag.com/archives2001.shtml
11. G.Dhillon and S.Moores. Computer crimes: theorizing about the enemy within. Computers & Security 20 (8):715-723.
12. H.Hacigumus, B.Iyer, C.Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. *2002 ACM SIGMOD Conference on Management of Data, Jun, 2002.*
13. E.Damiani, S.C.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.
14. E.Goh, H.Shacham, N.Modadugu, and D.Boneh, "SiRiUS: Securing remote untrusted storage," in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003.
15. 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.
16. E.Zadok, I.Badulescu and A.Shender. Cryptfs: A Stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.
17. S.Shepler, B.Callaghan, D.Robinson, R.Thurlow, C.Beame, M. Eisler and D. Noveck. NFS version 4 protocol. RFC 3530, April 2003.
18. A.D.S.G.Cattaneo, L. Catuogno and P.Persiano. Design and implementation of a transperant cryptographic file system for UNIX. In FREENIX Track: 2001 Usenix annual technical conference, June 2001.