Providing Data Sharing As a Service

Ravi Chandra Jammalamadaka†, Roberto Gamboni†, Sharad Mehrotra†,
Kent E.Seamons‡, Nalini Venkatasubramanian†

Donald Bren School of Information and Sciences†
University of California,Irvine
{rjammala,gamboni,sharad,nalini}@ics.uci.edu

Computer Science Department‡
Brigham Young University
seamons@cs.byu.edu

September 27, 2007

Abstract

This paper presents *DataVault*, an architecture designed for Web users that allows them to securely access their data from any machine connected to the Internet and also lets them selectively share their data with trusted peers. The DataVault architecture is built on the outsourced database model (ODB), where clients/users outsource their database to a remote service provider that provides data management services such backup, recovery, transportability and data sharing. In DataVault, the service provider is untrusted. DataVault utilizes a novel PKI infrastructure and encrypted storage model that allows data sharing to take place via an untrusted server. The confidentiality and integrity of the user's data is preserved using cryptographic techniques so that the service provider manages encrypted data without having access to the content.

1 Introduction

Modern day computer users generates a lot of personal information by utilizing computer applications. The proliferation of networking technologies and multimedia devices has served as a catalyst for such a trend. Examples of personal information that users generate include email, pictures, passwords, video albums, tax documents, work related documents, telephone directories, etc. Such personal information is scattered across diverse platforms and software.

Given that personal information is being produced, a natural questions to ask is, *can personal data be accessed from anywhere and anytime*. For instance, consider a user's address book which stores information about acquaintances. The user could generate the address book information from his/her home machine and desire access to it from the office machine.

When it comes to personal information another requirement is the ability to share it. Users share their personal information using a variety of methods such as disseminating information via email, posting data on a publicly accessible websites, etc. Such solutions have severe security drawbacks as unauthorized recipients can gain secure access to personal data. Expecting users to install and administer data sharing architectures is unrealistic and infeasible.

Imagine a file system on the Web that allows users to outsource their information to a service provider. The service provider can now provide services on the top of outsourced information and provide interfaces for the user to access such services from any machine connected to the Web, thereby achieving *device independence*. Also, the service provider can provide data sharing functionality for users to share data with other peers.

Such architectures already exist and are functioning commercially as well. For instance, Yahoo! Briefcase and Apple's Idisk are examples of such services. These services provide the user with disk storage and some preliminary data sharing functionality. Advantages of such services include: a) *Availability*: The data can be accessed 24/7 from any computer connected to the Internet; b) *relatively low cost*: The service provider can amortize the cost over several clients; and c) *Better service:* The service providers typically employ experts to provide better quality service. We refer to this as an *Outsourced File System (OFS)* architecture.

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

- **Outsider attacks:** There is always a possibility of Internet thieves/hackers breaking into the service provider's system and stealing or corrupting the user's data.
- **Insider attacks:** Malicious employees of the service 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 the attacks are insider attacks [2, 1].

Despite these security concerns, OFS architectures are gaining popularity due to the convenience and usefulness of the data services they offer. A user

of such a service can access his/her personal information from any computer equipped with a browser and an Internet connection. Besides mobility, the architecture lets users share their information with other users on the Web.

In this paper, we consider the problem of designing an OFS service when the service provider is untrusted. We describe the DataVault system that offers the same functionality as existing OFS systems with an untrusted service provider. This architecture is based on a realistic model that addresses the attacks that are prevalent today.

There are many challenges to address before an architecture of this kind can be built. The first set of challenges occur due to the requirement for designing a system that is *easy to use*. In other words, one of the fundamental tenets of DataVault is that it should be built keeping an average user in mind. To use DataVault, the user has to remember only one secret called the *master password* that controls the overall security that is provided by DataVault.

The second set of challenges occur due to requirements for preserving data confidentiality and integrity of client's data. To preserve data confidentiality, encryption offers a natural solution. The user's data can be encrypted before being outsourced to the server. When access to the data is required, the appropriate data can be fetched from the server and decrypted locally on a trusted machine. When encrypting user's data, care should be taken that the encrypted storage at the server side does not reveal information. In DataVault, we propose a *novel encrypted storage model* that does not reveal the content, metadata and the structure of user's data. The storage model that we propose also has performance benefits, which is described in the coming sections. DataVault also utilizes cryptographic techniques to detect if the data stored at the server is tampered with (i.e. ensures data integrity).

The third set of challenges occur due to data sharing requirements. In DataVault, data sharing is achieved at the level of access control. Access control on file systems is a well studied problem [16]. Most of the previous work largely concentrated on enforcing access control via a trusted server [17]. The trusted server is in charge of authentication and distributing information to the authorized recipients of the data. In DataVault, the server is untrusted and hence we cannot apply the techniques proposed previously. We explore an access control model that is *cryptographically enforced*. Data is encrypted and stored at the untrusted server. Both read and write permissions are granted by distributing the appropriate keys to the intended recipients. The problem then largely becomes of key management. Previous solutions on securing remote storage [7, 8] pushed the burden of key management to the user and assumed the presence of a trusted PKI infrastructure already in place. Such a model has the following inherent drawbacks: a) users have increasingly found it difficult to manage public/private key pairs; and b)

there are monetary costs involved in purchasing a public private key pair, thereby making it a less attractive proposition.

On the other hand, a PKI infrastructure is essential for enabling data sharing. Such an infrastructure allows data owners to share data with recipients if s/he know their public keys. Without a PKI infrastructure, the only other alternative is for the data owner to negotiate a secret symmetric key with every data recipient. This further places more burden of key management on the user. To combat the above issues, we introduce a *novel PKI infrastructure that is run collaboratively both by the client and server and pushes much of the burden of key management to the server.* Our PKI infrastructure is designed in such a way such that the server cannot compromise the security of the user. The user for all practical purposes is not aware of this infrastructure running in the background. This infrastructure, makes DataVault easy to use and yet retain most of the security properties offered by commercial PKI infrastructures. The reader should note that the scope of our PKI infrastructure is limited, it can only be used in our context and is not designed for use in other settings.

Cryptographically enforcing access control on file systems has been explored before [18, 19]. These models allow the data owner to grant privileges to recipients only at the level of *files*. This could be limiting in situations where the recipients also want read/write access at the level of directories. For instance, consider two groups of researchers placed geographically apart working collaboratively on an academic paper. We will assume the contents of this paper are placed in a directory, which further contains some directories that store the source code, tex documents, and figures. Let us assume that one of the groups wants to create new figures or add new tex files. Such a situation will require the groups to manipulate even the directory structure as well. To combat the above issues, we propose a *new cryptographic access control model that allows data owners to grant read and write privileges at the level of directories as well.*

Contributions: In summary, our contributions in this paper are as follows:

- We propose a complete design of DataVault architecture that allows data sharing via an untrusted server. The DataVault architecture is designed keeping an average user in mind. The user has to remember only one secret called the master password to use DataVault.
- We propose a cryptographic access control model that allows data owners to grant privileges both at the level of files and directories.
- We propose a novel transparent PKI infrastructure that is run collaboratively with the client and server and provides most of the function-

ality of commercial PKI services.

- We implemented a prototype of DataVault to measure the feasibility of such a service. A beta version of DataVault is available for download [20].

Roadmap: In section 2, we present a brief overview of our architecture. In section 3, we describe our PKI infrastructure. In section 4, we describe our encrypted storage model. In section 5, we show the different client server interactions. In section 6, we present the performance results of our working prototype. In section 7, we describe our related work and in section 8, we conclude.

2 Architectural Overview

Fig 1 illustrates our overall architecture.

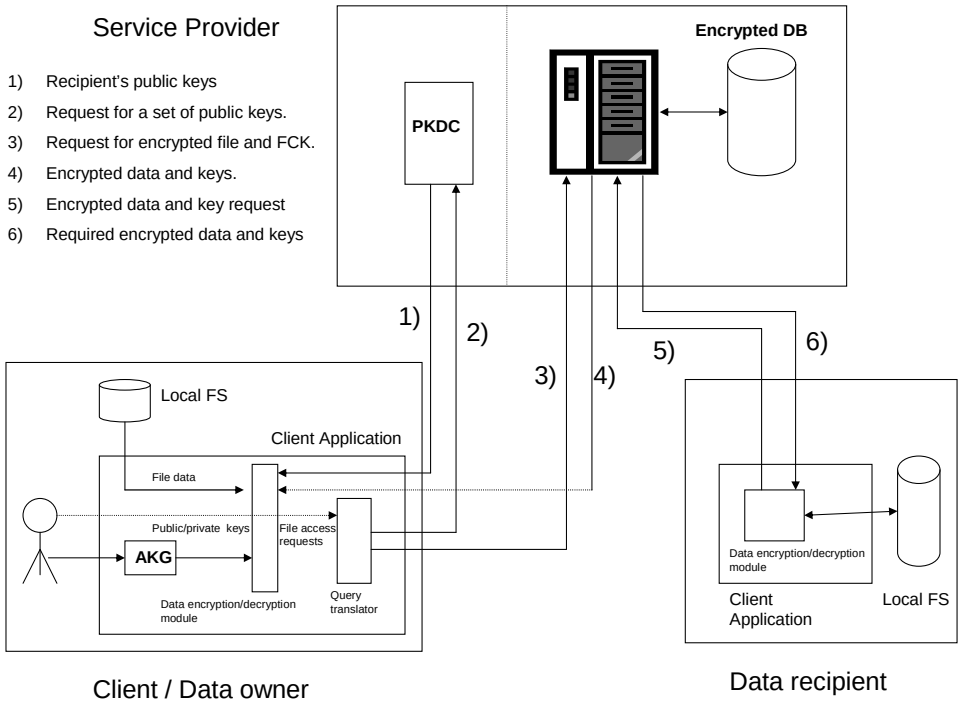

Figure 1: DataVault's architecture

Client/Data Owner: The data owner is the entity that produces and owns the personal data. The data owner is assumed to have some computational resources[1] but not comparable to the service provider. The data

[1]Any modern PC could serve as the client.

owner/client in DataVault is assumed to be a non-sophisticated user who is largely incapable of making security decisions. The client is also assumed to remember at most one secret to use DataVault, known as the *master password*. It needs to be kept completely confidential, since if an adversary gets hold of the master password, he can impersonate the client. The client side computing resources are assumed to be trusted (i.e., not under control of any malicious entities).

The client outsources local file system to the service provider. Before outsourcing, the client encrypts the file system to enforce its security constraints and access control policies. The keys used to encrypt the files are encrypted with the public keys of the user to distribute data. These public keys are provided by the server.

Server/Service Provider: The service provider is the entity in charge of providing data management services such as storage, backup, recovery, software updates, support for multi-modal access, etc., to the data owner. The service provider provides interfaces to the data owner to create, store and access their data. The data owners relinquish their administrative duties to the service provider. For reasons explained before, the service provider is untrusted. For purpose of this paper, we will adopt an *honest-but-curious* trust model for the service provider. The service provider or the corporation responsible for the service is expected to perform the services it advertises but any malicious employees are expected to be curious about data belonging to data owners and will try to profit by stealing such data. The *honest-but-curious* model has also been considered by previous data outsourcing approaches [3, 4].

As stated before, clients outsource their data in the form of files. In DataVault, the service provider stores files using a relational database management system (RDBMS). Alternatively, the service provider could have used a file system to provide the storage and retrieval for client's data. Database systems offer the following advantages over files systems: a) better multi-user support; b) better data searching capabilities and c) higher level of abstraction than files making development of DataVault easy. For these precise reasons, databases are getting very popular for maintaining data on the web. Utilizing a database as a backend for information management also saved us a lot of implementation time, since most of code for accepting user connections and managing them is already implemented in the databases. The server also maintains a public key distribution center(PKDC). The PKDC provides the required public key of the users and will be discussed in section 3.

Data Recipient: The data recipient could either be the data owner or another client of the service provider who has access to some of the data

owner's data. The data recipient can only access data according to the *data sharing* policy of the data owner. For instance, the data owner can allow a colleague access to work related documents. Data sharing is explained in more detail in the coming sections.

Usage Model: DataVault runs as an application on the client's machine. It provides a file system like view to the user. The user is not aware of the communication with server, to the best of user's knowledge s/he is accessing a local drive. The user can right click on a file or directory and share files with other users.

3 Public Key Infrastructure

This section describes a modest public key infrastructure to support the goals for DataVault. The PKI does not require some of the features of a full-fledged PKI, such as certificates, certifying authorities, and revocation. In DataVault, the client is responsible for generating public keys and the server is responsible for providing a repository for storing and retrieving public keys.

3.1 Client side PKI

DataVault requires that the user remember only one secret, the master password. This section shows how the client maps a master password to a public/private key pair. The reason for generating the key pair based on a password and not protecting a random key pair with a password instead is to allow the client to use any machine to generate the key pair without communicating with a server. Also, the mapping procedure always generates the same key pair from a given password. The client can then use the key pair to preserve data confidentiality and share files.

The key pair is generated using an *asymetric key genenator (AKG)* function.

Definition 1: *Asymmetric key pair generator(AKG) function* $F : P \rightarrow \{U_{pu}, U_{pk}\}$ *translates a secret string* P *into asymmetric key pair* U_{pu}, U_{pk}.

The asymmetric key generator function should have the following properties:

P1: Let E *be and* D *be asymmetric key cipher functions denoting encryption and decryption functions respectively. For any plaintext message* x,

$D_{pk}(E_{pu}(x)) = x$ *and* $D_{pu}(E_{pk}(x)) = x$.

P2: Function F should be deterministic. That is given a secret master password P, the function should always produce the same $\{U_{pu}, U_{pk}\}$.

P3: The cost of function F should be sufficient to deter a dictionary attack but efficient enough for a user to re-compute for each DataVault session.

Property P1 is satisfied using an appropriate public key cipher like RSA[26]. Property P2 is necessary so a user's password is always mapped to the same public/private key pair. Otherwise, the user will not be able to access her encrypted data.

Property P3 is important to prevent off-line dictionary attacks. For instance, since the asymmetric key generator function is public, an adversary with sufficient computing resources can try to guess a password and verify if it is correct in order to impersonate the user. Let c be the cost an adversary pays to guess a user's master password and run the asymmetric key generator function. If the adversary needs an average of n attempts to successfully reach user's master password, then the cost paid by the adversary is equal to $n \times c$. If n is sufficiently large, then we can relax the requirement for property P3. Since most users pick weak passwords, which can easily be broken using dictionary attacks, it is desirable to make c as large as possible to raise the cost of a dictionary attack. However, this is an interesting tradeoff since if c is too large, then the user is unnecessarily delayed before she can start using the system.

Our goal is to define an asymmetric key generator function that satisfies all the properties stated above. Our approach is to modify the traditional RSA key generation algorithm shown in Fig 2. The RSA key generation algorithm begins in step 1 by randomly selecting two large prime numbers p and q. Fig 3 shows a common method for accomplishing this by generating large random numbers and checking if they are prime using the Rabin_Miller test[26]. If the random number fails the test, another random number is generated and the process continues.

Fig 4 illustrates our AKG function, based on a modification of the prime number generation phase in RSA to make the algorithm more expensive. The algorithm utilizes a secure one-way hash function h. In order to generate the first prime number p, the master password P is repeatedly hashed k times to generate a number R. If R is even, it is incremented to make it odd. Then the primality of R is then checked. If R passes the test, the value of R is used as the prime number p. If not, then R set to $R + 2$ and checked again for primality. The process continues until a prime number is found. The

Step 1. Choose two large prime numbers p and q.
Step 2. Compute $n = p * q$.
Step 3. Compute $\phi(n) = (p-1) * (q-1)$
Step 4. Choose e coprime to $\phi(n)$
Step 5. Compute d such that $de - 1$ is divisible by $\phi(n)$
Step 6. (e,n) is the public key and (d,n) is the private key.

Figure 2: RSA key generation algorithm

1. R = Generate_Random().
2. if (! isOdd(R)) R++;
3. If (Rabin_Miller(r) == Prime) then {p = R} else
   Go to Step 1.
4. Repeat Steps 1 to 3 for q.

Figure 3: Picking large prime numbers

generation of prime number q uses the same approach, and begins by taking the value R and hashing it k times.

The hash iteration parameter k is a tunable parameter and plays a significant role in increasing the cost of the the asymmetric function. The adversary now will be forced to run the hash functions k times for every guess during a dictionary attack.

Care should be taken when choosing the hashing algorithm, since the size of the output determines the length of the key pair. In our implementation, we use the SHA-512 function to produce 512 bit output from the hash function. This implies that prime numbers p and q that we pick are 512 bits, hence the AKG generates a 1024 bit public key.

Cost analysis: Let c_i represent the cost of executing step i in the RSA algorithm illustrated in fig 2. The total cost of one run for the RSA algorithm is $c_{rsa} = c_1 + c_2 + \ldots + c_6$. Let c_{AKG} be the cost of our asymmetric key generation function (AKG). Then $c_{AKG} = 2 \times k \times c_h + c_{rsa} - c_1$ since we have replaced the step 1 of RSA algorithm with our repeated hashes of the master password P. Therefore, the slowdown of the RSA algorithm is equal to:

$$\frac{(2 \times k \times c_h + c_{rsa} - c_1)}{c_{rsa}}$$

Increasing the value of k increases the cost of the AKG function. Typically costs c_i where $i = \{1, 2 \ldots 6\}$ and c_h are very small in the order of

Input: Master Password P,
Hash iteration parameter k.

Function:

Step 1. Compute $R = h_k(h_{k-1}(h_{k-2}(.....h_2(h_1(P)))))$
Step 2. If (!isOdd(R)) R++ ;
Step 3. If (Rabin_Miller(R) == Prime) then {p = R} else {
R = R+2 and Goto Step 3}
Step 4. Compute $R^{'} = h^{'}_k(h^{'}_{k-1}(h^{'}_{k-2}(.....h^{'}_2(h^{'}_1(R)))))$
Step 5. If (!isOdd($R^{'}$)) $R^{'}$++;
Step 6. If (Rabin_Miller($R^{'}$ == Prime) then {p = $R^{'}$} else {
$R^{'} = R^{'}$ +2 and Goto Step 6}
Step 7. Follow steps 2 to 6 from RSA algorithm

Figure 4: Asymmetric key generator function

microseconds/nanoseconds, k is typically on the order of hundreds of thousands to increase the cost of the AKG to the order of seconds. If the client is willing to wait a few seconds while logging in, the AKG function will provide increased resistance to a dictionary attack.

Claim 3.1 *AKG function is a one way function.*

Proof Sketch: *The asymmetric key generation function first maps a master password MP to two different random numbers R and $R^{'}$ by iteratively hashing the master password. Let $h^i(x)$ represent a function that iteratively applies a secure hash function such as SHA512 on a string. Since SHA512 is a one way hash function, $h^i(x)$ is also a one way hash function. The AKG, then maps $< R, R^{'} >$ to $< U_{pu}, U_{pr} >$ pair using the RSA key generation function. It is well know that RSA function is a one way function. Since the AKG internally utilizes two one way functions, the adversary cannot determine the value of master password MP, when provided with the $< U_{pu}, U_{pr} >$ pair. Hence, the AKG function is a one way function.*

Claim 3.2 *AKG function is a secure function.*

Proof Sketch: *We will say the AKG function is insecure, if an adversary $\mathcal{A}$ can obtain the private key U_{pr} of a user U without trying all possible master passwords.*

From theorem 3.1 we know that AKG is a one way function. Therefore by knowing a public key $< U_{pu} >$ of a user U, $\mathcal{A}$ cannot determine the master password MP. Since we map a master password to a set of random numbers R and $R^{'}$ used for the public/private key generation, $\mathcal{A}$ can potentially try

the whole space of random numbers in a brute force manner to determine the private key U_{pr}. The output of SHA512 or the random numbers, is 512 bits long. That implies that A on the worst case needs to try 2^{512} possible random values to determine the private key. In the average case, $\mathcal{A}$ needs to try $2^{512}/2$ possible random numbers, which is a very huge number.

The only option that is left for $\mathcal{A}$ is to try all possible master passwords in a brute force manner. In DataVault, the master passwords needs to be at least 8 characters long. The master passwords should at least contain 2 special characters, 2 numbers and 2 upper case characters. The user can of course choose a password that is more than 8 characters long. Let us assume that the user barely satisfies the requirements[2]. Number of possible master passwords are:

$$8! \times 27^4 \times 30^2 \times 10^2 = 1928493100800000$$

In DataVault, the iteration parameter is set to a value that ensures that AKG function takes about 1 min. Therefore the adversary potentially can spend more than 3669126904 years trying to break the AKG function. Hence, the AKG function is secure.

3.2 Server Side PKI

The server is responsible for providing access to the user's public keys. If a data owner wants to share a file with a data recipient, he/she requires access to the recipient's public key. To achieve this, all users register their public keys generated using the asymmetric generator function with the *Public key distribution center (PKDC)*. The PKDC then provides access to the public keys of other users to the requestor. Note, if two users choose the same passwords, then their public keys will be the same. This is clearly undesirable, as both users could have access to each other's data. To prevent such a situation, DataVault needs to ensure that no two users pick the same public/private key pair. This is achieved in the following manner: In DataVault, each user is identified by a unique username. The username is actually the email address of the user. The server ensures that during registration, no two users register under the same email address. The server also ensures that the user registering to the server is the valid holder of the email address, by sending an email with a random token to the address. Now only the valid user with access to the email account can provide the token back to the server and hence authenticate himself/herself. Once the uniqueness

[2]Unfortunately, in practice we have found the users to pick minimally satisfying passwords.

of the username is ensured, the user is free to choose any password. The *masterpassword* is derived in the following manner:

Masterpassword = Hash(Username, User_chosen_pasword)

Where *Hash* is any strong cryptographic hash function such as SHA384, SHA512, etc. Since the username is unique, any collision resistant hash function output of the username and the chosen password of the user, will always be unique. The derived master password will now be used to generate the public/private keys using the asymmetric generator function described in section 3.1. Now, no two users will have the same public key. The PKDC stores the <username, public_key > pairs for all users. Note, by using the email address as the username, we have ensured to an extent that no malicious user assumes a false identity. Most current online users identify other online individuals with email addresses and this design naturally fits with such a model.

Note that if the PKDC does not function properly, then there could be a possible loss of data confidentiality to the user. For instance, if the PKDC behaves maliciously and provides the wrong public key to a user, then the user might end up revealing it's information to an adversary. To prevent such things from happening in DataVault, the PKDC is run in a separate administrative domain from that of the database server. The service provider should make sure that there is no common employee that works both at the database server side and the PKDC. Such a measure, we will show later provides tremendous security to the user. This assumption of non-colluding servers collaboratively providing a service has been previously been made in [14]. In that work, the authors provide techniques to store and maintain data objects with two non-colluding servers. Unless the servers collaborate the value of the data object will not be revealed.

4 Encrypted Storage Model

This section describes the server side representation of user's data. The design of the encrypted storage model(ESM) should have the following desirable properties:

- **Hide structure:** ESM should not leak information about the file system content, metadata and the structure. 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 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. Also, the standard for XML encryption proposed by the W3C consortium also hides the structure of the XML documents [15].

- **Access control:** ESM should enforce the access control policy of the user. Access control policies in DataVault are cryptographically enforced. Further details about the access control enforcement is described later in the section.
- **Dynamic nature:** ESM should be amenable to the dynamic nature of the user's file system. The effort to encrypt/decrypt a file should be minimal when the user updates the file system.

The following definitions are necessary to understand our model.

Definition 2: File System: *A user's file system is represented as a graph* $G =< V, E >$*, where* V *is a set of vertices that represents both the file and directory nodes and* E *represents the set of edges between them. 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 represent 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.*

Definition 3: File Structure: *Let the graph* $G =< V, E >$ *represent the File system of a user. The file structure of the file system is also a graph* $G' =< V', E' >$*, where* V' *inherits all the nodes from* V*. That is* $\forall n$*, where* $n \in V$*,* n *also belongs to* V'*, but the nodes content* $n.content$ *is set to null.* E *and E' are equivalent sets, which we will represent as* $E \equiv E'$*.*

The file structure is very central to our model. It contains the nodes on which users specify the access control primitives.

Definition 4: Access Control Model: *The access control model of a user is a set of rules denoted by* $R = \{ r_i : 1 \leq i \leq n\}$*. Each rule* r_i *is a tuple of the form* $< node, R_e, W_r >$ *where* node *represents a node in the file structure,* R_e *represents a set of users with read access and* W_r *represents a set of users with write access.*

Notice that our representation of the access control rule is a more compact form of the popular <subject, object, access> representation. As we will see later, the expressiveness of both representations is the same. Consider the rule R_1: $< n_1$, {John, Jake}, {Bob, Alice} $>$. This rule states that John and Jake have read access to node n_1 and Bob and Alice have write access to the

node. We will henceforth use the notation $n.Read$ and $n.Write$ to represent the set of users with read and write access for node n.

In DataVault, the data owner can navigate to a directory or a file, press the share button and specify the set of *readers* and *writers* to the file. This process of the owner specifying the access control policies on the internal nodes of the file structure can be termed as *annotation*. The user by annotation transforms a file structure F to an *annotated file structure*. The nodes that have access control policies specified on them are called *annotated nodes*. For instance, in fig 5a, the node n_1 is an annotated node.

We will now describe the semantics of the access control rules in DataVault.

Semantics: When a user is provided access to an internal node it implies access to all nodes underneath it. This property is the *containment property* that is inherently present in access control models for file systems. When the user is given access to a directory, it is implied that the user has access to the directories and the files underneath it.

In DataVault, there is no strict write access. Users are granted *Read-Write* access. That is, for every user U granted write access to node n, U also has read access to node n. Such semantics are necessary for many data sharing applications.

Currently, we do not allow negative authorizations which exclusively prevent an user from accessing certain nodes in our model. In future, we plan to extend our access control semantics in this direction.

4.1 Rewrite Rules

We do not place any restrictions on the user when specifying access control rules. The user can identify any node and specify any arbitrary number of readers and writers. Due to such flexibility, the resulting annotated file structure could posses redundant or inconsistent information in it. We will describe some rewrite rules that remove such redundancies and inconsistencies.

Read-write inconsistency: Consider the file structure represented in fig 5a. For a node n_1, a user $\mathcal{U}$ has specified an access control rule $< R, W >$[3]. Let us assume that there exists a user U_1 such that $U_1 \in W$ and $U_1 \notin R$. This violates the semantics that we described above, where every user with write access also acquires read access immediately. In situations, we rewrite the contents of R to contain users represented by $R \cup \{n : n \in W \text{ and } n \notin R\}$.

Read redundancy: Consider the file structure represented by the fig 5b). Consider two annotated nodes n_1 and n_2 where $n_1 \leftarrow Ancestor(n_2)$. We

[3]We have ignored the node in the rule for simplicity

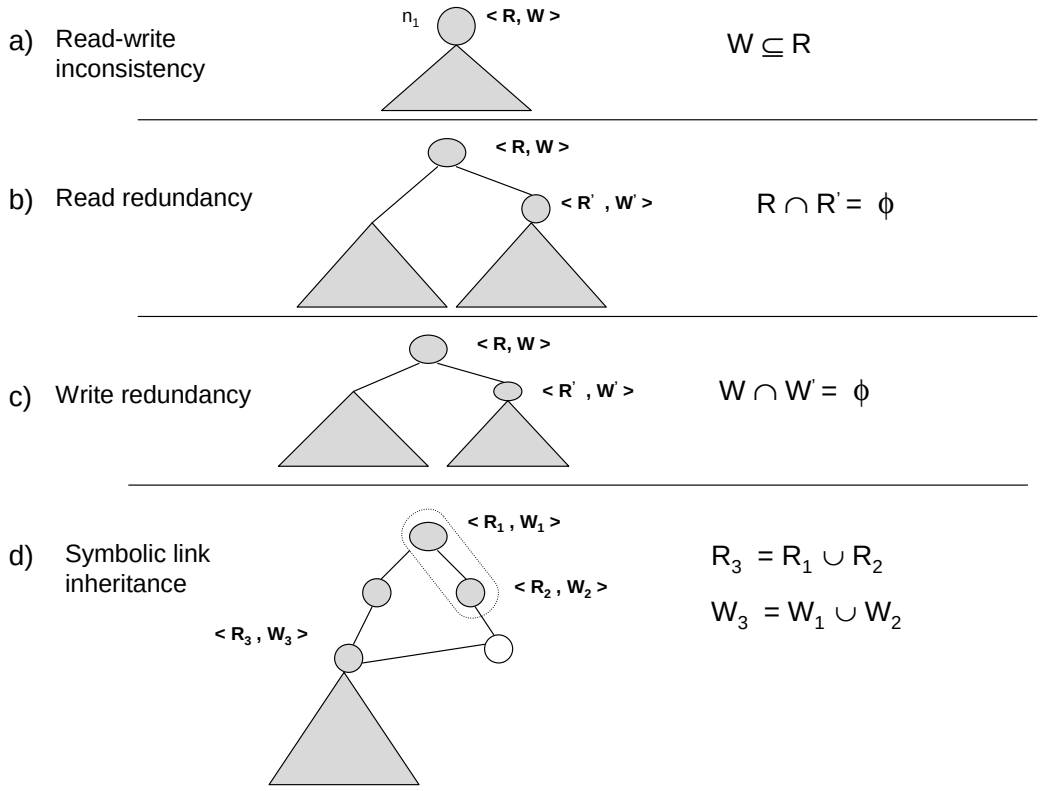

Figure 5: Rewrite rules

will assume that $n_1.Read \cap n_2.Read \neq \phi$. That is, $\exists r$, such that $r \in n_1.Read$ and $r \in n_2.Read$. Since the user r has read access to node n_1, due to the containment property r has read access to n_2. There is no requirement for user r to be part of the set $n_2.Read$. Removing this redundancy has performance benefits that will be clear soon. In such situations, we remove the user r from $n_2.Read$. We ensure that no two nodes which belong to a single path, contain a user in common in their readers set.

Write redundancy: This rewrite rule follows the same pattern for the writers set that was described above for readers.

Symbolic link inheritance: Consider the file structure represented by fig 5d). The unshaded node in the figure represents a symbolic link. When a user access the symbolic link, s/he is requesting access to the node the symbolic link points to. Therefore, all the users that have access to the symbolic link, should have access to the node the symbolic link references. Let a node s be a symbolic link referencing node n_1. Let the function $Ancestor(s)$ represent all the ancestor nodes of node s. Then, $\forall v$, where $v \in Ancestor(s)$, node n_1 inherits all the users with read and write access. That is,

$$n_1.Read \equiv n_1.Read \bigcup_{\forall v \in Ancestor(s)} v.Read \quad (1)$$

$$n_1.Write \equiv n_1.Write \bigcup_{\forall v \in Ancestor(s)} v.Write \quad (2)$$

Given a file structure, the rewrite rules are evaluated in the order we presented them. Following the rewriting process, we say the file structure is *well formed*.

4.2 Enforcing access control cryptographically

In this section we describe how a well formed annotated file structure is encrypted to satisfy the access control rules the user specifies. We will first start by providing the intuition of our approach before we formally present it.

Intuition: Let us assume that a data owner wants to share a data object with a set of recipients. A data object could be a file or a directory. To provide read access to a data object, the data owner encrypts the object with a *object encryption key(OEK)* randomly generated. This object encryption key is then distributed to the recipient. The details of how this is achieved are explained in section 5. The recipient can now decrypt the object and read the object contents. To provide write access, the data owner generates a object integrity public/private key pair $< OIK_{pu}, OIK_{pr} >$. The OIK_{pu} is publicly known. That is all users of DataVault have access to the key. The data owner distributes OIK_{pr} along with the OEK to recipients whom s/he desires to have write access. The recipients when they want to make changes to the data object can encrypt the updated object with the OEK, calculate the integrity information of the object and encrypt the integrity information with OIK_{pr}. In DataVault, we use the HMAC function to calculate the integrity information of the encrypted object. The value of the HMAC function is stored along with encrypted object. Now any user of DataVault can verify that the *changes are correct* by decrypting the encrypted integrity information using OIK_{pu}. We will use the notation, OEK_{n_1} to represent the object encryption key of node n_1. Similar notation is used for the object integrity public private key pair.

Basic Technique: Consider the example well formed annotated file structure represented in fig 6. There are two annotated nodes n_1 and n_2. To satisfy the access control requirements of node n_1, we need to encrypt the complete $subgraph(n_1)$ including the files underneath it. The function $subgraph(n_1)$ represents the subgraph rooted at n_1. This would interfere

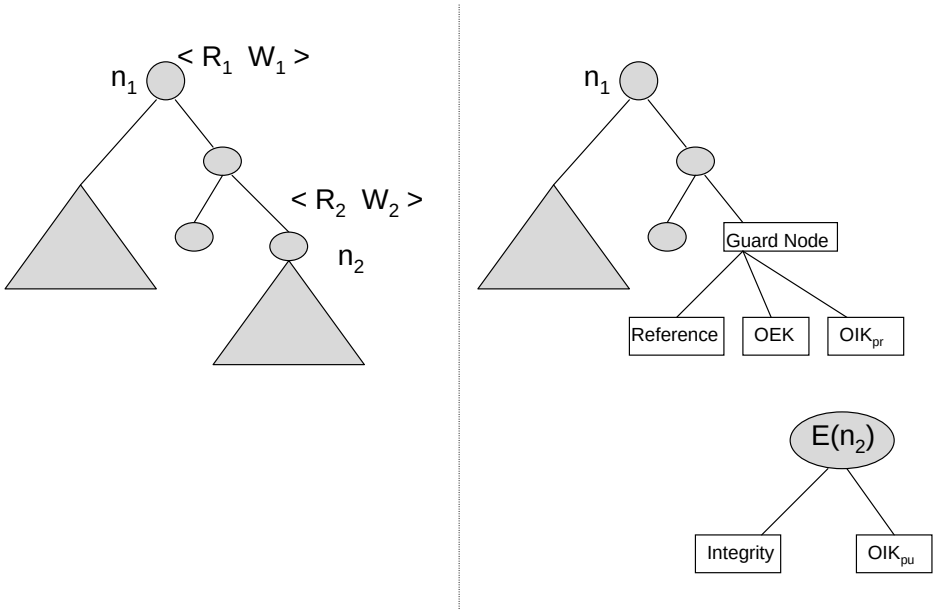

Figure 6: Rule enforcement

with the access control requirements of n_2, since the read and write set of node n_1 differs from node n_2. If the path from n_1 to n_2, does not contain another annotated node, then n_1 and n_2 are *immediate annotated nodes*.

Given two immediate annotated nodes n_1 and n_2. Let us assume as in the figure that height of node n_2 is greater than that of n_1. We first isolate $subgraph(n_2)$. $subgraph(n_2)$ can now be considered as an data object. The data owner generates an OEK_{n_2} and $< OIK_{pu_{n_2}}, OIK_{pr_{n_2}} >$ for the object/node n_2. The owner then encrypts the object with the OEK_{n_2}. Which means that the owner encrypts the file structure and the nodes underneath the node n_2. This encrypted object is represented as $E(n_2)$ in fig 6. The owner computes the integrity information of the $E(n_2)$ and signs it with OIK_{pr}. Then, a *guard node* whose structure is shown in fig 6 is added to the parent node of n_2. The guard node is an ancillary node that DataVault adds to the file structure. It is completely transparent to the user. The guard node contains the OEK_{n_2} , $OIK_{pr_{n_2}}$ of node n_2 and a reference node that points to $E(n_2)$. The contents of the reference node are explained later in the section. The OIK_{pr} of node n_2 is encrypted with OIK_{pr} of node n_1, which implies that a user having write access to node n_1 will also have write access to node n_2. We then repeat the procedure we performed on node n_2 on n_1. That is we generate OEK_{n_1} and $< OIK_{pu_{n_1}}, OIK_{pr_{n_1}} >$, encrypt the $subgraph(n_1)$ with OEK_{n_1}, calculate the integrity information and place it

```
Input: An annotated file structure F_S
BEGIN:
        h = MaximumHeightofAnnotatedNodes();
        while( h > 0) {
        for all annotated nodes at height h {
                /* Let n_1 be the current node.
                Find its immediate annotated ancestor node n_2.
                if there is no immediate annotated ancestor node {
                /* we have reached the root node.
                        Encrypt the subgraph(n_1) to compute E(n_1)
                        Compute the integrity information for E(n_1)
                } else {
                        Encrypt the subgraph(n_2) to compute E(n_2)
                        Compute the integrity information for E(n_2)
                        Add the guard node to parent of n_2
                       }
              }
        h = h -1;
        }
END:
```

Figure 7: Basic Scheme for access control enforcement

under the $E(n_1)$ node.

After the encryption process, all the users in $n_1.Read$ are distributed the key OEK_{n_1} and all users with write access in $n_1.Write$ are distributed $OIK_{pr_{n_1}}$. Similarly, we do the key distribution for node n_2. Hence, access control is achieved via key distribution.

Basic Scheme: Fig 7 illustrates our overall basic scheme for cryptographically enforcing access control in file systems. We first find annotated nodes with the maximum height and their corresponding immediate annotated node. Then, the procedure described above is executed. This process is repeated until there are no more annotated nodes. In the algorithm described in fig 7, we need a quick way to find annotated nodes with the maximum height. To achieve this, we maintain a priority queue along with the file structure. The priority queue maintains pointers to the nodes in the file structure. This gives us a quick way to access the required nodes. Notice that in enforcing access control we are breaking a file system into a set of encrypted subgraphs. In the remainder of this paper, we will refer to subgraphs as *file system chunks* or *chunks* for short.

Observation 1: *Let the number of annotated nodes in a file structure be n_a. Let n_k be the minimum number of object encryption keys (OEKs) required to satisfy the access control requirements according to the basic scheme. Then $n_a = n_k$.*

Observation 1 states the minimum number of encryption keys required

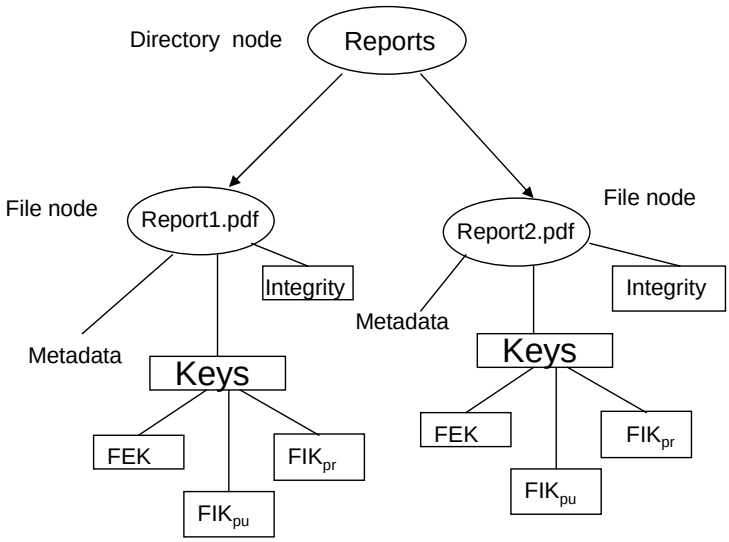

Figure 8: Final Scheme

for satisfying the read access requirements of the access control rules. Observation 1 can also be easily extended for write access requirements of the access control rules as well. It is quite straightforward to see why this is the case, since for every annotated node we are generating one object encryption key and one object integrity public private key pair.

The basic scheme has one drawback in terms of security. Annotated nodes with maximum height contain the content of the files. All these files are encrypted with the OEK of the annotated node. Such a scheme could generate a large amount of ciphertext encrypted with the same key. Particularly, if the subtree of the annontated nodes contains a large number of files. This could potentially make the data vulnerable to *Cryptanalysis attacks* whose effectiveness increases with the size of the ciphertext. We will now present the final scheme that counters such attacks.

Final Scheme: To enforce the access control rules, we split the file system into a set of file system chunks. Each chunk, contains a set of files as leaf nodes. In our final scheme we separate the files from each chunks. Each file is individually encrypted using the a file encryption key (FEK). This is done to counter cryptanalysis attacks by reducing the amount of ciphertext encrypted with one key. We also generate a file integrity public/private key pair $< FIK_{pu}, FIK_{pr} >$ for generating and verifying the integrity informa-

tion. These keys are stored in the file structure as shown in the fig 4.2. We then follow all the steps described in the basic scheme. The user who has access to the annotated node, will also have access to the files underneath the node, since s/he can access the file keys. Finally we store the set of chunks and files in the following relational tables [4].

CHUNK(ID, ENCRYPTED_CONTENTS, INTEGRITY);
FILES(ID,ENCRYPTED_CONTENTS, INTEGRITY);

For every chunk, DataVault assigns a unique id calculated as follows: *id = h (username, full_path_of_root_node)*, where h is a one way hash function. The username is used to ensure the uniqueness property of the ids. Recall that we introduced *reference nodes* that refer to chunks. The value of the reference node is the *id* of the chunk it is referring to.

The file ids are calculated a little differently. The id of a file is calculated as follows: *fileid = h (username, filename, random_number)*. We do not use the path of the file when generating a file id, since it can change when the file is moved. Instead, we use the file name and a random number that is generated when the file is created. The random number is used to ensure that no two files with the same name map to the same id. The random number and the file id are placed in the chunks to which the files belong.

4.3 Updates to the access control policy

We will discuss how updates to the access control policy are handled in DataVault. There are two situations that can occur when an access control policy is updated.

When annotation to a node is changed: Consider an annotated node n whose read and write set is represented by $< R, W >$. The owner can make changes to both the R and W either by adding users or removing them. When users are added to these sets, the required keys OEK or $< OIK_{pu}, OIK_{pr} >$ are distributed to the users.

Revocation, i.e., removing users from R and W is a bit tricky, since malicious users can continue to cache the keys that were distributed to them. We therefore need to re-encrypt all the chunks and their associated files that are underneath the node n with a new set of keys and distribute the set of keys to authorized recipients. This could potentially be an expensive operation, depending on the number of chunks that need re-encryption.[5] In

[4]Recall that the server maintains an RDBMS for storing and retrieving data.

[5]In practice we have found that it is not usually an expensive operation, since number of chunks to re-encrypt usually tends to be less than or equal to two.

DataVault, the chunks that are effected by revocations are flagged at the server to prevent other users from modifying them, and a separate thread running at the data owner side with low priority, re-encrypts the required data. This allows the data owner to continue to work while the data is being re-encrypted.

New annotation: Let n_1 be the new node that is annotated by the data owner. We find its corresponding immediate annotated node n_2 and apply the basic technique that we described before. This will split the chunk rooted at n_2 into two chunks, one rooted at n_2 and one rooted at n_1.

4.4 Security Discussion

In this section we will discuss the relevant security issues in DataVault.

Information revealed: 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 number of chunks of the user's file system and their relative sizes:* All the file structure chunks are downloaded by the user as soon s/he logs in. The size of the file structure chunk linearly increases with the number of nodes inside it. Hence, the adversary can reasonably guess the number of directory and file nodes in each chunk. The adversary does not get any further information regarding the user's file system.

Another alternative for the encrypted storage model is to create a data object that subsumes both the file structure chunks 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.

Inherit trust on recipients: Access control in DataVault is achieved via key distribution. The server stores the encrypted data and provides access to the data to any user of DataVault. Hence, a malicious data recipient can provide read/write access to other users by sending the relevant keys to them. DataVault currently assumes that all data recipients are trusted and do indulge in such attacks. Our future work will deal with preventing such miscue of access. One possible solution would be to push some part of the access control to the server. The server can take advantage of the access control mechanisms of the database to provide the required functionality.

Such a study is outside the scope of this paper. Previous work on securing remote storage [7, 8] also placed similar trust on the recipients.

5 Client-Server Interaction

This section describes the various client-server interactions present in the DataVault architecture.

5.1 At Login

When the user first logs into DataVault, the complete file structure of the user is retrieved. The server maintains the root file structure chunk for every user. After the root chunk is retrieved and decrypted, the client applications will follow the *reference nodes* (see section 4) and fetch all the chunks that belong to the file system automatically.

By fetching the complete file structure, the client application can answer all user navigational requests locally. This improves the usability of the system, as the user does not have to wait repeatedly while navigating the file system. Typically, the complete file structure can be fetched in less than a second. The file structure is fetched in a separate thread, while the client application is mapping the master password to public/private key pair using the AKG function. The AKG function is deliberately slowed down to 1 min, and during this time the file structure is fetched. Another advantage of maintaining the file structure as a set of chunks, is that it saves time in decryption. If we were to follow an NFS based storage model [21], where the files and directories are stored as separate data objects at the server, we would be forced to decrypt each of the nodes individually, instead of doing it in bulk. This saves us time as most encryption algorithms have a huge startup time.

5.2 Updates to the file system

One of the drawbacks of our storage model is that handling updates to the file structure is a bit tricky. Updates operations in DataVault include creating a directory, renaming a directory, creating a file, renaming a file, etc. We will explain updates in DataVault under two situations.

Updates to the unshared part of the file system: Consider a situation where the user creates a new directory. The file structure can be updated locally to reflect the change. To make the change persistent we need to visit the server and store the updated file structure. Doing this for every update

is expensive. To solve the above problems we use an approach similar in style to the journaling file systems. Whenever an update is made to the file system, only the update is stored at the server in the form of a log. If the application crashes at the user side, when it reboots, it will check if there any log entries at the server side. If there are, then it will apply the updates to the file structure. If there are no crashes, then after the user logs off, the file structure is updated at the server side and the log removed. All the log entries are encrypted at the server for security purposes. We have not shown the language for representing the updates in the interest of space, we hope that the above discussion provides the reader with enough intuition for the entire procedure.

When the user is updating or creating a file, the contents of the file are not written to the log, instead a message is inserted into the log that a file creation or an update is under progress. The contents of the files are updated immediately at the server side. If the upload of the file contents fails due to a software crash, then after the application comes back up, DataVault will look at the log and alert the user. It is up to the user to take the necessary action.

Updates that involve annotated nodes: Updates that involve annotated nodes are handled immediately. That is we do not delay the update till the user logs off. Let n_1 be a annotated node in the user's file system. Let us assume that a data owner or a recipient with write access wants to create a file in the $subgraph(n_1)$ region. To execute this operation, the relevant chunk is fetched from the server and the file node is added at the relevant place. After the update operation, the chunk is encrypted and it's integrity information is calculated. The chunk is then updated at the server.

5.3 Data Sharing

We will now provide some details of how data sharing is achieved in DataVault. We will discuss this under the following two situations.

Read Sharing: When the data owner O wants to provide read access to $subgraph(n)$ of some node n to a user U, O distributes OEK_n to U. The OEK_n is encrypted with U_{pu} the public key of user U to generate $E_{U_{pu}}(OEK_n)$ which is stored at the server. The public key is fetched from the PKDC. Now, when required U can fetch $E_{U_{pu}}(OEK_n)$ from the server and decrypt it with its private key to reveal OEK_n.

The DataVault interface separates the files that are shared to the user from the local file system. All the shared files are shown in a different tab labeled as *incoming files*. When the user clicks the incoming files tab, all the chunks shared to the user are fetched and decrypted. The files that belong

to the chunks are not fetched at this stage. The user can then navigate the chunks as she would navigate their local drive. When the user clicks on a file, the encrypted file is downloaded and decrypted using its FEK, which is present in the chunk. The *readers* of a chunk/file will always read the last committed changes to the chunk/file from an authorized writer.

Write Sharing: When a data owner O wants to provide write access to $subgraph(n)$ of some node n to a user U, O distributes OIK_{pr} to U. This is done by encrypting the OIK_{Pr} by the public key of user U. The result $E_{U_{pu}}(OIK_{pr})$ is stored at the server. When U desires access, it will fetch $E_{U_{pu}}(OIK_{pr})$ from the server and decrypt it with its private key to reveal OIK_{pr}

As stated earlier, in DataVault there is no write access, only read-write access. As in read access, all the subgraphs to which the U has write access are shown in incoming tab. The interface separates the subgraphs to which the users has read access from the subgraphs to which the user has write access. When the user wants to modify a subgraph, she needs to be acquire a *lock* from the server. The lock is a flag that is set at the server. This is done to prevent data inconsistencies. Only after the user acquires the lock, the server will allow the updates to propagate.

6 Performance

We have developed an initial prototype of DataVault based on the design and the techniques described in this paper. We conducted experiments to measure the performance of the prototype. This section provides the implementation details and some of the performance results of DataVault. For more detailed experimental results, please refer to the full version of our paper available at [20].

Implementation details and experimental setup: The client side interface for DataVault was implemented using the JAVA 5 SDK. We plan to map this interface to an applet and distribute such an applet via a web server. This will allow the users to visit a website and access their files stored in DataVault. Currently, the users have to download the *jar* file containing all the classes and execute the program locally. All the cryptographic operations were implemented using the Java Security API, which contained most of the cryptographic techniques required in this architecture. At the server side, we used a MYSQL database to store and retrieve encrypted content. The MYSQL server was running on a powerful 32 GB RAM, 8 processor machine. We implemented the PKDC as another MYSQL database engine running on a different machine. The PKDC was run on a P4, 2.66GHZ machine. Note,

File Size in MB	Network Time in secs	Encryption /Decryption in secs	Integrity Calculation in secs	Database Insertion in secs	Database Retrieval in secs
0.5	5.12	0.140	0.094	0.235	0.500
1	10.24	0.172	0.109	0.172	0.187
2	20.48	0.219	0.125	0.438	0.813
5	51.2	0.375	0.203	1.093	1.422
10	102.4	0.594	0.328	2.250	1.406
15	153.6	0.781	0.469	2.906	1.891
18	184.32	0.953	0.531	3.250	7.797

Figure 9: Comparison of cryptographic overhead, network overhead and database overhead

while we assumed the service provider as one logical entity, in reality the service provider could maintain a host of machines for both the PKDC and database server. This will make the server not vulnerable to single of points of failure.

We conducted some experiments to measure the cost of the techniques proposed in this model to determine if the system design is practical. User's will not tolerate lengthy delays for basic operations in DataVault. This section demonstrates that the system is practical.

Performance Results: The purpose of the experiments was to measure the various delays caused by our techniques. The delays primarily include a) *Network delays* caused due to transferring files over the Internet to the server; b) *cryptographic delays* caused due to the encryption/decryption of the files and the data signatures computation; and c) delays caused due to storing and retrieving files using a database, which we will refer to as *database delays*. Although, database delays should not be considered as delays, since we need mechanisms to store and fetch data, we nevertheless measure the time taken to store and fetch files from a database since the user has to wait for that amount of time. During the experiments we have assumed the network transfer speed between the client and the server to be 800 kilo bits per second(kbps). This speed is much less than what we were getting in the laboratory, since the server was in close proximity to our client machine. The 800kbps assumption is made to simulate clients that are geographically far from the server. Fig 9 states all the delays that are caused due to data outsourcing in DataVault. The Encryption/Decryption column includes the cost of encrypting the file using a symmetric key (FCK) and encrypting the symmetric key with an asymmetric key (client's public key). The integrity

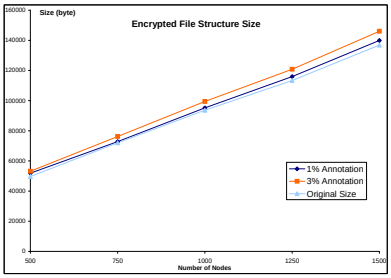

Figure 10: Encrypted file structure size

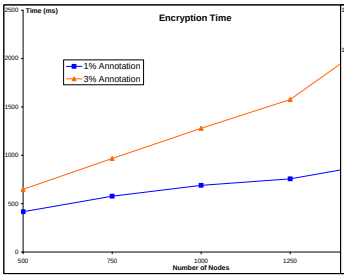

Figure 11: Encryption time

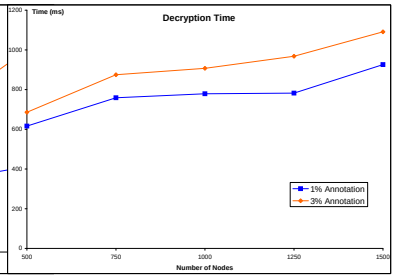

Figure 12: Decryption time

column states the cost of computing the message digest of the file and signing it with file/object integrity private key. We varied the sizes of the files being outsourced to measure the impact of file size in the delays. The values that we report are the average of a few attempts of transferring and receiving files from the server. Fig 9 shows that *cryptographic and database overhead is negligible when compared to the network overhead which is unavoidable*. This implies that the client is not paying much of a cost for security. The network cost is unavoidable for the user if he/she wants the service. This was an encouraging result for us since most users are patient with downloading and uploading files.

We also wanted to measure the overhead of our cryptographic access control model. Due to the specifications of the annotations, the file system structure is broken into a set of encrypted file chunks that are stored at the server. We wanted to measure: a) the size of the encrypted file chunks; b) the time required to enforce the access control as specified by the annotations; and c) the time required to obtain the plaintext form of the file structure from encrypted chunks. For these experiments, we used a local file system of one of the authors.

Size of the encrypted file chunks: Due to the specifications of annotations, the size encrypted file chunks tend to be larger than the plaintext file system structure. This is due to the injection of the guard nodes which carry the required encryption keys. Fig 10 plots the increase in such a size when 1% and 3% nodes [6] are annotated. As it can be seen from the figure, the increase is minimal and therefore, the user does not have to pay a significant amount of network cost in transferring the encrypted chunks to the server side.

Enforcing access control: Fig 11 plots the time required to enforce the access control specified by the annotations. Even when 3% of the nodes

[6]Most users do not share a large part of their file system

Operation	Time in secs
Total Login Cost	1.5
AKG function	1.0590
Reading the file structure	0.0084205
Decryption of the file structure	0.0008687
Initialization of the file structure	0.4095802
Checking if the masterpassword is correct	0.0231397

Figure 13: Login Cost

Operation	Time in secs
Adding a file	0.0142464
Deleting a file	0.0001290
Deleting a Directory	0.0006919
Creating a Directory	0.0001998

Figure 14: Operations to the internal file structure

are annotated for a fairly large file system, it takes only about 2 secs for the enforcement. This is a delay that most users are unlikely to notice.

Decryption of the file chunks: When the data owner first logs in, he/she fetches all the file chunks, decrypt them locally and stich them back to original file structure. Fig 12 shows the time required when 1% and 3% of the nodes are annotated. This process in the worst case takes less than 2 secs.

In the next experiment we tried to measure the total time the user needs to wait before he/she can log into the server. We will term this time as the login cost. Please refer to the section 5 to see the various steps involved in logging in. Fig 13 states the respective costs involved. The client has to wait for 1.5 secs to login to the DataVault. This is primarily due to our

Operation	Time in secs
Encryption of the Structure	0.0008687
Storing the structure	0.0080949

Figure 15: Costs paid by the client during logoff

deliberate slowing down of the asymmetric key generation function to achieve added security. We feel waiting 1.5 secs should be acceptable for most users. Although, this time can be tuned down for individual tastes albeit with a reduction in security. After the file structure is fetched from the server it is instantiated as a XML document locally. We then measured the time taken by different file system operations that manipulate the file structure or the XML documents. Fig 14 illustrates the performance costs for some of the file system operations. We did not report the cost for all the file system operations, since they followed the similar pattern. As it can be seen from the figure, such costs take negligible amount of time and the user is unlikely to notice the time. Finally, we measured the time taken for the user to log off from the DataVault service. Fig 15 shows the results. In keeping with the above experiments, the time taken is negligible and unlikely to be noticed by the user. In summary, the experimental results show that there is negligible overhead due to cryptographic and database operations, which makes this architecture practical.

In summary, the cryptographic costs of our technique pale in comparison to the network costs that are unavoidable. Also, the network speeds are improving and that will decrease the time the user needs to wait. Therefore, we believe that DataVault is a feasible practical architecture.

7 Related Work

The two research areas that come close to our work are cryptographic file systems and database as a service architectures.

Cryptographic file systems [7, 8, 10, 11] provide file management to users when the underlying storage is untrusted. This is typically the case when data is stored at remote untrusted servers. Two examples of cryptographic file systems related to our work are Sirius [7] and Plutus [8]. Sirius layers a security mechanism on an underlying file system, insisting that no changes be made to it. The goal of Sirius is to provide the user with data confidentiality and integrity when the data is stored at an untrusted storage. Plutus file system is also based on the goals of Sirius. Additionally, Plutus introduces the concepts of lazy revocation and novel group sharing strategies.

The fundamental differences between our work and cryptographic file systems previously proposed are: a) cryptographic files systems build a security layer over legacy systems. Therefore no changes can be made to the untrusted server; b) the systems were built for sophisticated users; c) access control model is limited only to the level of files. For instance, in Plutus the users were expected to do all the key management. In Sirius, the users are

expected to purchase a public/private key pair and securely transport it when mobile access is desired. We are not concerned with legacy systems. We are exploring/building a service architecture that allows data sharing requirements of the clients to be outsourced. Hence, while the previous solutions insisted on no changes to the server, in DataVault we design a server that is optimized for data sharing. Our architecture is catered to average computer users and is easy to use. We also propose a new access control model that allows the data owner to grant privileges at the level of directories as well.

DAS [3, 4] 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. We are concerned with providing mobility and data sharing to personal information mainly belonging to individuals, although our architecture is not restricted only to individuals and can be easily extended to organizational usage as well. Mobility and data sharing were not handled in DAS, which is the primary focus of this work.

Miklau et. al. [12] propose a framework for enabling access control over published data. In their model, the data owner encrypts the relevant parts of an XML document in such a fashion that all the access control requirements are preserved. The data owner publishes the information to a public server and by key distribution access control is enforced. In their model, the untrusted model is just responsible for storage and the data owner is left to do a lot of work, primarily key distribution. In many respects this architecture is similar to Plutus, but deals with a different data model. On similar lines, Bertino et. al. [24] propose an architecture to publish XML documents securely via third party servers. This work mainly concentrated on exploiting a Merkle tree approach to ensure data integrity.

Jungle disk [25] is a commercial software that layers a security mechanism over the Amazon S3 storage 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. We cannot evaluate the encrypted storage model of the Jungle disk, as there is no documentation available. The data sharing model is also not as advanced as DataVault.

8 Conclusions

We have introduced and presented the design of DataVault, an architecture that provides users mobile access to their personal data and also allows them to share their data with other peers on the Web. DataVault is catered to average users who are are not cognizant of security technologies and are largely incapable of making security related decisions. Users have to remember only one password to use DataVault, that controls the overall security provided by DataVault. DataVault is based on a outsourced database model (ODB) where the data is outsourced to a remote service provider. Since the service provider is untrusted, clients encrypt the data before outsourcing. The service provider now provides data services over the top of encrypted data. We proposed novel cryptographic access control model and a PKI infrastructure that allow data sharing to take place via an untrusted server. We have implemented an initial version of DataVault and measured its performance, much to our satisfaction. A beta version of DataVault is available for download [20].

As part of future work, we plan to incorporate search techniques on encrypted data into DataVault [22, 23]. This will allow the user to fetch all the required files which contain a particular keyword. In the current avatar, DataVault 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 DataVault to outsource such files, then DataVault 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 DataVault.

References

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

[2] G. Dhillon and S. Moores. 2001. Computer crimes: theorizing about the enemy within. Computers & Security 20 (8):715-723.

[3] 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.*

[4] 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.

[5] S. Ross, J. Hill, M. Chen, A. Joseph, D. Culler, E. Brewer. A Composable Framework for Secure Multi-Modal Access to Internet Services from Post-PC Devices, Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications, December 2000. Page(s):171 - 182

[6] J.Jeff, Y. Alan, B.Ross, and A. Alasdair. The memoribility and security of passwords - some empirical results, 2000.

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

[8] 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.

[9] J.Shanmugasundaram, K.Tufte, C.Zhang, G.He, D. J. DeWitt, J. F. Naughton: Relational Databases for Querying XML Documents: Limitations and Opportunities.*International Conference on Very Large Databases (VLDB 1999): 302-314*

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

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

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

[13] D. Balfanz, G. Durfee, and D. K. Smetters. Making the Impossible Easy: Usable PKI. In Security and Usability, L. Crannor and S. Garfinkel, editors. Chapter 16, pp. 319–333. O'Reilly Media, Inc., August 2005

[14] 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.

[15] XML Encryption Syntax and Processing. http://www.w3.org/TR/xmlenc-core/.

[16] A.Silberschatz, P.Baer, G.G.Gagne. Operating System Concepts. John Wiley and Sons, Inc. ISBN 0-471-69466-5.

[17] J.P.Anderson. Computer security planning study. Technical Report 73-51, Air Force Electronic System Division, 1972. Computer Science, 1997.

[18] A.Harrington, C.Jensen. Cryptographic access control in a distributed file system. Symposium on Access Control Models and Technologies, 2003.

[19] V. Kher and Y. Kim. Securing Distributed Storage: Challenges, Techniques, and Systems . In Proceedings of the first ACM International Workshop on Storage Security and Survivability (StorageSS 05).

[20] The DataVault project. http://www.ics.uci.edu/ ˜rjammala/DataVault/.

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

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

[23] E.J.Goh. Secure Indexes. In submission.

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

[25] http://www.JungleDisk.com

[26] Alfred J. Menezes, Scott A. Vanstone, Paul C. Van Oorschot. Handbook of Applied Cryptography. CRC Press, Inc.