Encrypted File System

6.858 Final Project

Cheng Chen & Osama Badar

Abstract

We built a userspace file system with minimum server support that allows efficient sharing of files among users and achieves fork consistency. Our filesystem allows us to share files with multiple users while making sure that all operations are secure. We achieve filesystem security using a combination of data encryption and merkle tree.

1.0 Introduction

To achieve privacy, we encrypt everything. To achieve integrity, we do Merkle tree, i.e. recursive hash. Now it's secure but not efficient so we use different keys for different inodes. In order to share a file, we hand this key to the user. If we revoke the permission, we re-encrypt all the way to the root, however there is no need to traverse the children. This is because what the attacker can get is unchanged files that he already has access to. To achieve this, we need that each update operation will use a new key to encrypt that inode, and therefore updates to the root. However, since parent node stores the key and hash of the children node, we need to remove this dependency by making shared node a new root, and join it back to the original tree when no one is sharing it. This should be taken care carefully to make sure that other user can still visit that shared node from its parent node. Indepently, we want to make the server as minimum as we can since that means we can achieve security with minimum server support. To do this, we first let our file name to be in the range of some hash function, therefore we can do the read/write permissions. It might worthwhile to mention that filename is changed every time we update a file, except those related to the root. Also, old files should be removed from the server. The last thing is to do is fork consistency, that is by storing all the operation history, including both read and write, in the server, and the client does the hash by himself. Hash is also done in a manner that depends on the timestamp and all the former operations, therefore the server cannot cheat at any given timestamp. To do the check, clients sharing files run the state command to update the consistency file and output current timestamp and hash value.

1.0 File System Design

The underlying file system is based on a merkle tree where a root node depicts the root directory, leaf nodes depict files and the rest of the nodes depict directories. Directories and files are stored as i-nodes on the server. An i-node on the server contains encrypted information. All file names are converted to random hashed strings and all data is encrypted using a symmetric key encryption scheme, for example AES + CBC. This ensures that the server cannot read file

names or data on the server because the client saves the root decryption information locally in a special file called ‘profile’ that can be used to decrypt the data recursively down to the required i-node.

We achieve privacy of data by encrypting it using symmetric encryption scheme. This guards the data against a malicious adversary or server who can access data but will not be able to decrypt it since the symmetric key used to encrypt this data is stored locally by the client. We also ensure that our file names are encrypted in parent directory hence allowing privacy. We also aim to establish data integrity by using merkle trees on the filesystem. A merkle tree applies recursive hash to all its nodes starting from the leaves to the root. For example, a root node is the hash of the labels of its children’s nodes and this applies recursively to all nodes. Merkle trees are a useful data structure as they allow efficient and secure verification of data. In our implementation we use SHA1 because of its efficiency.

Figure 1 shows a typical i-node. For example, a directory i-node contains information about its children that can be either directories or files. The information it contains is the original name of the file/dir mapped to some encrypted information about that file. This includes its encrypted file name, hash of encrypted file name, private key to decrypt file data and the hash of the symmetric key in that order. The tree is hashed all the way up to the root to ensure that the files are not tampered with by a malicious server. If a server corrupts the data inside a random directory, it will be detected during integrity checks by the client while he uses his locally stored symmetric key to decrypt the root. A corrupted directory means that the root hash would not match and the file fails the integrity check.

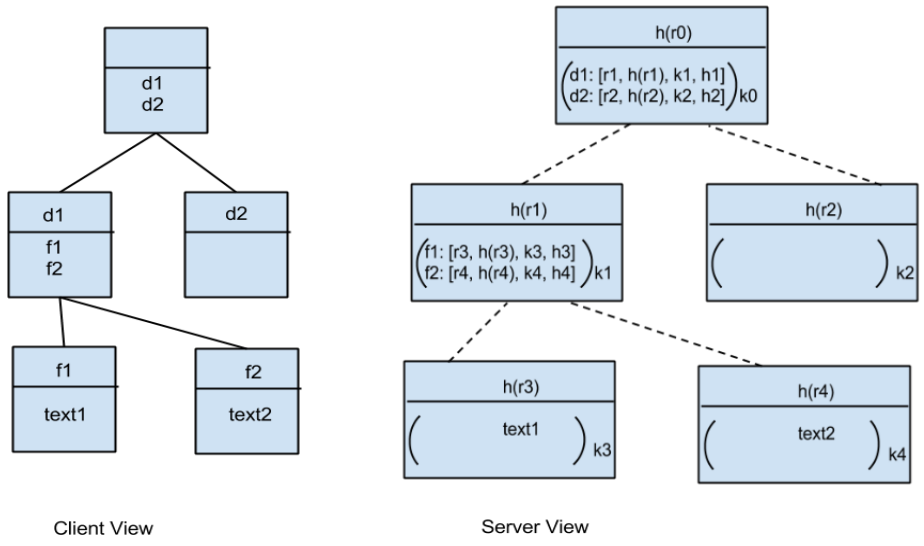

Figure 1: File system structure. The left tree is the client’s view of the file system while the right tree depicts

the same i-nodes on the server. Each file i-node contains data encrypted using a private key and each dir i-node contains the information of its child i-nodes that be used to decrypt the child nodes. The tree is hashed all the way up to the root.

2.0 Operations supported by File System

Our encrypted file system supports the standard unix file system operations such as ls,pwd,mkdir,cd,rm,rmdir,put/get file, and it also allows us to share files among users with read and write permissions. In the following sections, we only touch upon the interesting operations.

2.0.1 Write/Modify File

In order to write/modify file, the user calls method put(filename). We create a new i-node that is encrypted using a new key for the file that needs to be written to the server. Since a new hashed i-node is created, we update all the parents of the current directory in our merkle tree by recalculating all hash values based on the hash value of the node inserted. In order to achieve this we have to fetch all parent i-nodes from the server, update their hash values and resend them to the server. We also delete all the old i-nodes from the server in order to preserve space. Figure 2 explains how we update the merkle tree after each write.

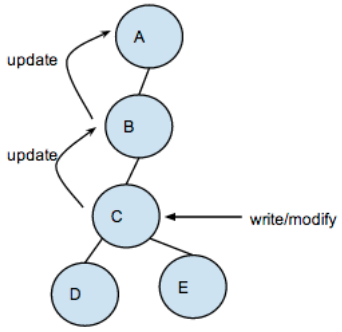

Figure 2: To write to a node C, fetch i-nodes C,B,A from server and delete them. Make a new i-node C, and re-encrypt all parent nodes of C to the root. Post these i-nodes back to the server.

2.0.2 Permissions

Our file system has two permissions namely read and write. The owner of the file has read and write permissions. To share a file with another user, a user can use read or write permissions and to revoke his permissions, he can use a permission of none. Permissions are stored in a separate file on the server and they are updated with a call to every filesystem command using updatePermission() call.

2.0.3 Share File

A user can share files with another user using the setacl(username,permission) command. Since each i-node in the system is encrypted using a unique key, a user can efficiently share a file with another user by providing him the unique symmetric key of that file through a secure channel. The shared user can read or write to this directory. However, this leads to some complications such as updating the entire tree structure every time a shared user makes changes to the file. Therefore, to share a file, it is first cut off from the tree and made into a new root. Both users have access to the symmetric key of the this newly created root because the root hash is stored in a consistency file on the server that both users can check.
Figure 3a explains this process in detail. Shared root C is cut off from root so any shared users only need to update their immediate parent when they modify the files.

In order to revoke permissions to a shared file, the owner of the file will re-encrypt the shared directory and all its parent nodes all the way to the root with new random key. Figure 3b explains revoke. Once the dir is re-encrypted,a shared user cannot access it anymore. We do not need to encrypt all the children nodes of C in figure 3b. This is because even if the shared user accesses these files, he is looking at old data which he originally had access to. If these files were modified , they would have been re-encrypted and the shared user would not be able to access them.

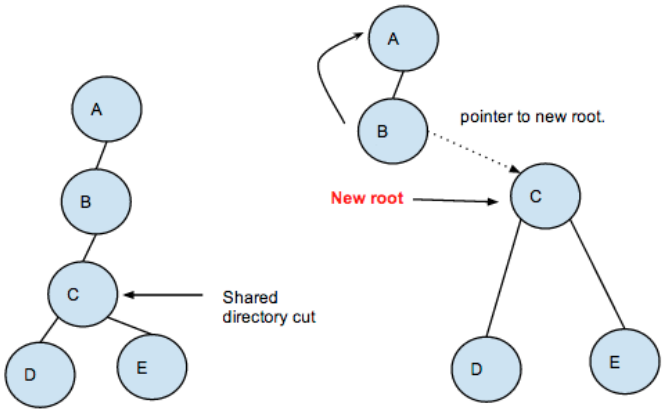

Figure 3a. Sharing a file. Cut the shared node to create a new i-node. Each user who intends to write to file C D or E only re-encrypts them using a unique key along with the root and the hashed value of root is stored in consistency file. Both users have access to consistency file.

Moreover, we only connect back the new root C to the original tree, if it is not shared by any user anymore.

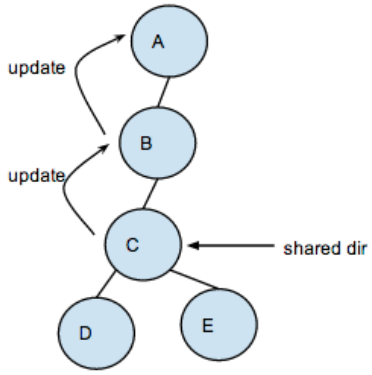

Figure 3b. Revoke a shared file. Node C is shared with another user. To revoke permissions, re-encrypt all parent nodes of C using new keys. This ensures, that the user does not have access to make changes to C again. All children nodes of C do not need to be encrypted since a write to them ensures that they are re-encrypted and if they are not, the shared user is looking at stale data. C is also joined back to the original tree is no user is sharing it.

There is a caveat here though. We realize that our choice to not re-encrypt the child nodes when we revoke permissions is efficient but it will not work if we have a shared directory under a shared directory. In that case, we also need to re-encrypt the child nodes all the way down to the child directory which is also shared. There are two ways to solve this issue. We can disable the option to have shared directory within a shared directory because it is easily detectable thereby avoiding this all together. Another more efficient way is that we can maintain another tree for all the shared directories and find any child shared directory with minimum overhead . At any given time, we can find all the shared directories under a given shared directory and re-encrypt all children till that shared directory.

3.0 Consistency Check

Our file system provides fork consistency that can be verified using the state() command of the filesystem. The user calls updateconsistency() with any filesystem command that logs the action with a time stamp as shown in figure 2. This hash value can be used to verify that the state any given time is consistent. The invariant is that if everything is consistent, then for every t, dt are the same for every user. If the server ever cheats by performing replay attack or giving any outdated data, since read operations are also recorded, the clients can detect it. The history is

stored in the server, but the hash is not, and is computed by user.

	time	user	Old hash	New hash
r0	0	u1	None	h0
r1	1	u2	h0	h1
r2	2	u2	h1	h2
r3	3	u1	h2	h3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
rn	n	u1	hn-1	hn

d0=hash(r0)
d1=hash(r1, d0)
d2=hash(r2, d1)
d3=hash(r3, d2)
.
.
.
dn=hash(rn,dn-1)

Server **Client**

Figure 4. Server and client side views are shown. Any activity on the server is logged using a timestamp.Client can make sure that the data is consistent at all times using the hash of the previous d0 value as shown.

4.0 Conclusions

We designed and implemented an encrypted file system that relies on a merkle tree for integrity checks and ensures users privacy using standard encryption primitives. Our filesystem not only supports basic unix file commands but it also allows us to share files with multiple users efficiently. We believe our minimum server functionality coupled with our encryption and integrity checks protect our filesystem from a malicious adversary over the network and a malicious server.