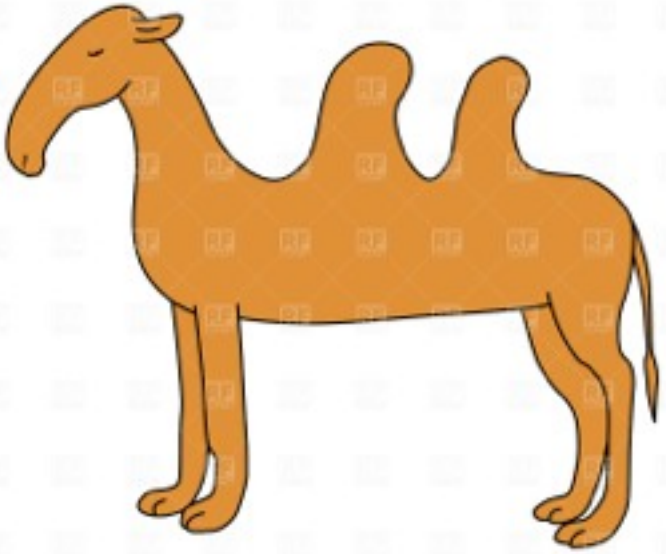
CAMELFS

An Encrypted File System to Secure Storage on Untrusted Remote Servers



E. Sila Sayan, Eric Soderstrom, Tiffany Tang
{erensila, e_k_s, fable}@mit.edu
6.858 Fall 2013

CAMELFS

An Encrypted File System to Secure Storage on Untrusted Remote Servers

E.Sila Sayan, Eric Soderstrom, Tiffany Tang
6.858 Fall 2013

Abstract

CamelFS is an encrypted file system which allows users to store files on an untrusted server without revealing file contents or names. The system also enables file sharing in read-only or read-write permission modes. Some interesting design aspects are 1) Access control is decentralized by implementing permissions as capabilities attached to the resources themselves (i.e. carried in file meta-data), 2) A two-tier security approach which enlists the help of the server in case it is cooperative and trustworthy, but doesn't *depend* on it.

System Goals

(See Appendix A for full list of requirements system satisfies)

• Minimize number of keys that users must keep track of

• Two-tier Security : Don't depend on an untrusted server for security, but allow system to leverage server's help if it is trusted.

Our Design Process

We knew from the start that given an untrusted server, we would want to design permissions more like capabilities rather than UNIX-like access control lists. After all, with a server that is not to be trusted, who would be in charge of enforcing the requirements of these access control lists. Further, we did not want to assume a trusted third party, because it seemed like that would be side-stepping the real problem.

To hide file names and contents from the server, we needed encryption. Clearly, the notion of capabilities in this case was going to be distilled into *cryptographic keys*. In particular, we observed the following natural correspondences between file permissions and security concepts:

Read permission <--> Data confidentiality --> Encryption

Write permission <--> Data integrity --> Signing

As such, a user with read permissions would need a decryption key, and a user with write permissions would need a signature key as well as a decryption key, to be able to provide a valid, verifiable signature upon modifying the file.

We went through several iterations of design as we tried to tame the beast that was the problem of key exchange: To achieve file-level granularity for sharing and permissions, there needed to be unique encryption and signature keys per file. We also needed unique keys per user. Furthermore, we wanted to have as little out-of-band communication as possible to simplify matters.

The idea we eventually settled upon had the following outline:

- Files carry their own associated encryption and signature keys (i.e. capabilities are attached to the resources themselves)
- Files carry their own permission information, which express which users have access to the file's read and/or write capabilities (i.e. no central access control table necessary)
- The permission information carried by the file is enforced cryptographically, without a central authority having to marshal authorization decisions.

We further refined this idea and fleshed out the specifics of how to cryptographically enforce permissions by referring to [1].

Another option for reducing the key management load could have been to take an approach similar to that in [2], where the logical grouping is built around files, not users. That is, [2] "groups files (not users)". We chose not to go this route, as permissions based on users, rather than *filegroups,* was a more familiar paradigm we found more natural and elegant to work with.

The Final Design

Table 1. Terminology

Term	High Level Definition
File	Contains the data that is to be put on the server/shared
File log (f-log)	Contains metadata about the file, most important of which is are "Access Blocks". Files and their corresponding f-logs are associated by the system.
Access Block	Data structure that contains permission information in the form of username and granted capabilities.
Directory log (d-log)	Contains metadata about the directory, much like an f-log. Directories and their corresponding d-logs are associated by the system.

Keys in the System

CamelFS has 3 unique keys per file, and 2 unique keys per user.

The File Encryption Key is an AES key which functions as a read-capability. The File Signature Key is a DSA key, which functions as write-capability. Though, since write-only permission doesn't make sense, if a user has the File Signature Key for a file, he/she must also have the File Encryption Key. The third file key, Block Encryption Key, is an AES key used for as an intermediary for encryption to solve some kinks in implementation. (See note below Table 3).

The User Encryption Key allows the keys for a file to be encrypted on a per user basis, such that a user may have access to a file key only if it has been encrypted with their User Encryption Key and included in the file log (file logs and files explained in next section).

Meanwhile, The User Signature Key allows the owner of a file to sign a file log, allowing receivers of this corresponding

Table 2. Unique Keys per User

Key	Description	Used For
User Encryption Key	RSA key pair	Encrypting per file keys meant for user in the the file log's access block
User Signature Key	DSA key pair	Signing file log if owner of file

Table 3. Unique Keys per File

Key	Description	Used For
File Encryption Key	AES key	Encrypting file contents
File Signature Key	DSA key pair	Signing hash of file contents
Block Encryption Key	AES key	Encrypting File Signature Key *

* Note: The Block Encryption Key was needed to make the implementation work. A File Signature Key needs to be signed by the RSA keys of those users who have write-permission to the corresponding file. We found in practice that the DSA key generated for file signatures was too long to be directly encrypted by an RSA key. Therefore, we have an intermediary AES key with which to first encrypt the File Signature Key, and then sign with appropriate users' User Encryption Key.

File Structure in the System and Additional Data Structures

File and f-log

Every file which contains data, also contains a hash of its contents signed by the File Signature Key.

For every file which contains some data, there's an associated f-log file which contains metadata.

The f-log: Information Content

It contains information about:

File owner

List of users with whom file was shared

File Signature Key (public portion)

Timestamp of last modification of f-log

Encrypted name of the associated file

For each user, including the owner, there is an Access Block which contains the:

File Encryption Key

File Signature Key : None if user doesn't have write permission

Block Signature Key: None if user doesn't have write permission (see Note below Table 3)

The f-log: Implementation

The f-log is a file which consists of a dictionary containing the above mentioned metadata, and a signature of this dictionary at the end (signed by the file owner).

Example.

Assume file *foo.txt* owned by Alice.

She wants to share this file:

read-only with Bob

read-write with Chris

Table 4 shows what information the dictionary in the corresponding f-log file, *foo.txt.flog,* contains.

Table 4. Contents of dictionary in foo.txt.flog

KEY	VALUE	COMMENTS
'owner'	'Alice'	
'Alice'	Access Block for Alice	by default, owner's Access Block always contains the File Encryption Key, File Signature Key, and Block Encryption Key, all encrypted with owner's User Encryption Key
'timestamp'	time f-log was last modified	
'encrypted_name'	string "foo.txt" encrypted with foo.txt's File Encryption Key	
'file_dsa_public'	Public portion of foo.txt's File Signature Key	
'users'	['bob', 'chris']	
'bob'	Access Block for Bob	Bob has read-only access. So his Access Block contains foo.txt's File Encryption Key encrypted with Bob's User Encryption Key
'chris'	Access Block for Chris	Chris has read-write access. So his Access Block contains foo.txt's File Encryption Key, File Signature Key, and Block Encryption Key, all encrypted with Chris's User Encryption Key

Lastly, *foo.txt.flog* contains a hash of the dictionary signed by Alice's User Signature Key.

Directory and d-log

Directories work just like they do in UNIX file systems. The directory and d-log relation is analogous to the file and f-log relation.

Access Block

An AccessBlock object contains a File Encryption Key, a File Signature Key and a Block Encryption Key. File Signature Key and Block Encryption Key are None if the object is created for a user who only has read permission on the particular file.

In the AccessBlock object, the keys are all encrypted by the User Encryption Key of the user for whom the the AccessBlock was created.

Key Exchange

The only keys that need to be exchanged out-of-band in our system are User Encryption and User Signature public keys. We assume that the users will use a PGP key server to exchange these keys.

Two-Tier Security

In case the server is not malicious, we would like to able to utilize it as a second layer of "safety blanket". As such, the server keeps two tables:

User table contains: Usernames, hashed passwords and salts

File table: For each file keeps track of users who have write permission, and users who have read permission

If the server is non-malicious and cooperative, it can stop an attacker without read and/or write permissions before the attacker tries to even read/write. Similarly, the server checks that only the owner of a file or directory can delete it.

But it's important to note that our system does not depend solely on this. The cryptographic system we have described so far makes sure that even with a malicious server which colludes with attackers or features an adversarial system administrator, the users can stop and/or detect attacks mentioned in the project specification (Appendix A).

Prevention/Detection of Malicious Acts

User without read permissions cannot decrypt the file to read it.

User without write permissions cannot produce valid signature for file.

Server cannot successfully use providing foo.txt when bar.txt is requested as an attack vector because a) it cannot see decrypted filenames, so when user gets file and decrypts name, user will see it was the wrong file, or b) if the server provides bar.txt.flog but foo.txt, the user won't be able to get the correct decryption key to read the file, so the attack will be useless.

Lastly, the system is set up such that all files in a user's home and subdirectories are owned by the user. So if the server maliciously tries to add a file to a user's directory it will be detected, as the server doesn't know the user's private key to be able impersonate him/her.

Limitations

CamelFS satisfies the requirements posed by the project specifications. There are, however, improvements that we would've hoped to make had we had more time. For instance, with the infrastructure that we've already built for CamelFS, it would be a natural next step to add f-log freshness files with a Merkle tree implementation to make freshness guarantees.

We also haven't explicitly implemented access revocation features, but our design makes it easy to do so: The owner would just need to update the f-log/d-log files and also re-encrypt the file/directory in question.

Conclusion

We implemented an encrypted file system that provides a Directory structure and basic UNIX file operations, while maintaining confidentiality of file/directory names/contents. The most interesting features of our design are that 1) We are approaching permissions as capabilities implemented as cryptographic keys, and attaching the capabilities to the resources themselves, 2) We are de-centralizing access-control by allowing each file/directory to contain its own permission information, and relying on cryptography to enforce the access control policy that these permissions define, 3) We are taking a two-tier approach to security in the face of a possibly malicious server, i.e. we are not depending on the server for security, but we are enlisting its help when we can.

Appendix A

Encrypted file system

Your goal for this project idea is to develop a file system that allows users to store data on an untrusted file server. The file server should not be able to obtain the user's plaintext data (i.e., your file system should encrypt the data), and the file server should not be able to corrupt the data either (i.e., your file system should authenticate the data it gets back from the file server).

- Your file system should support many users, and allow users to share files with one another. For each file, it should be possible to control the set of users who can read, and who can write, to that file.
- At a minimum, your file system should meet the following requirements:
- Users can create, delete, read, write, rename files.
- The file system should support directories, much like the Unix file system.
- Users should be able to set permissions on files and directories, which also requires that your file system be able to name users.
- File names (and directory names) should be treated as confidential.
- Users should not be able to modify files or directories without being detected, unless they are authorized to do so.
- If the server is not malicious, unauthorized users should not be able to corrupt the file system.
- The file server should not be able to read file content, file names, or directory names.
- The server should not be able to take data from one file and supply it in response to a client reading a different file, without being detected.
- A malicious file server should not be able to create or delete files or directories without being detected.

BIBLIOGRAPHY

[1] Goh et al SiRiUS: Securing Remote Untrusted Storage

[2] Kallahalla et al Plutus: Scalable Secure File Sharing on Untrusted Storage