CryptFS: A FUSE-Based Networked Secure File System

Beth Findley, Leonid Grinberg, Betsy Riley
{findley,leonidg,rileyb}@mit.edu

1 Introduction

We present CryptFS, a networked secure file system implemented in Python. CryptFS ensures that files stored on the server are encrypted, so that a malicious or compromised server cannot access their contents. Additionally, it maintains the privacy of file and directory names, and it prevents the server from tampering with file contents or supplying incorrect files to the client without detection. Finally, CryptFS maintains a rich permissions system so that a user can grant reading, writing, or administrative privileges on individual files or directories to other users.

CryptFS is implemented on top of FUSE, which allows it to be mounted as if it were a local filesystem. Nearly all standard UNIX commands will work on CryptFS, with the exception of those that manipulate file permissions, for which CryptFS provides a different tool, as well as directory renames.

2 Software requirements

CryptFS is implemented in Python (we used Python 2.6) and uses the PyCrypto[1] and fusepy[2] libraries. Additionally, data is stored in a SQLite3 database.

3 Architecture

3.1 Client-server communication

CryptFS is organized as a standard multithreaded client/server system. The server "hosts" a directory tree and provides RPC bindings that, for the most part, act as simple wrappers around POSIX filesystem calls. The client loads its RPC calls through FUSE and is thus able to access the directory tree hosted on the server.

[1] https://www.dlitz.net/software/pycrypto/
[2] https://www.dlitz.net/software/pycrypto/

3.2 Security

3.2.1 Privacy and integrity of files and filenames

The client maintains a table of AES keys for each file it has access to. A file's AES key is used to encrypt its contents before they are sent to the server and to decrypt the contents received from the server. The server is thus unable to read the contents of files. Directories also have AES keys, which are used to encrypt the names of the files inside of a directory; the server is thus also unable to learn the names of file paths. (The single exception to this is the root directory, which for convenience has a well-known AES key. Names of files directly inside the root directory are therefore not private.)

Additionally, to ensure that any tampering with the contents of files by a malicious server can be detected, the beginning of each file contains a SHA1 hash of the file's contents after AES decryption. The client checks that the hash of the contents matches the provided hash, and, if it does not, reports that the server has tampered with the file contents. Similarly, to prevent the server from responding with a different file from the requested one, a file—again, after AES decryption—begins with the non-encrypted path of the file. Since the server never sees the real names of files, it is unable to fake that line; the client checks that the line matches the path it is expecting to get and reports if it does not.[3]

3.2.2 User authentication

Each client acts on behalf of a "user," who is identified by a username. Users correspond to sets of AES keys, which are stored in a SQLite database by the client. A user "logs in" by decrypting her database with a password and writing to a special file that indicates the current user. A user "logs out" by encrypting her database with some password, which she should remember in order to decrypt upon her next login, and clearing the special file.

Besides the AES keys, each user's database also stores an RSA keypair. CryptFS comes with tools for exporting and importing public RSA keys, which users may share with each other and with the server. Each RPC call is tagged with the user's username and signed with the user's private key. The server verifies the signature and provides the verified usernames to each system call. This allows users to authenticate themselves to the server and provides the basis for CryptFS's file permission system.

This file permission system works through a permissions database stored on the server. Each row of the database contains an AES-encrypted path (i.e. the path known to the server), a username, and the unencrypted path and AES key for the file, encrypted as a pair with the public RSA key of the user. Lastly, each row contains a permission level ("read," "write," or "admin"), which different system calls can refer to when deciding whether to honor a particular command (e.g. `read()` will not work unless the user has at least "read" privileges, `write()` will not work unless the user has "write" or "admin" privileges). The

[3] An unfortunate consequence of this is that `rename()` cannot be implemented server-side. Instead, the client manually copies the contents of the file to the new location and deletes the old one. Of course, this only applies to renaming files; in fact, CryptFS does not support renaming directories (see footnote 4 in the following subsection).

server also provides a special RPC call for granting permissions to a user—a user may only grant permissions to a given file if he or she has "admin" privileges to that file, which are originally granted to the user who created the file.[4]

When a user attempts to access a file for which they do not have an AES key, the client will attempt to download the key from the server via another RPC call. Because the AES key and path are encrypted on the server with the user's public RSA key, the server can neither learn the AES key nor fake the path to which it points. This prevents a subtle security flaw in which a server could generate a fake AES key for a file that was being shared with a user and hope that the user uses that key to write to the file. At that point, not only would the contents of the file be visible to the server, but they would also be invisible to the original writer of the file, who would still be using the old AES key. Thus, a malicious server could essentially "piggy-back" off of another client to hijack a file and read its contents. The fact that the AES key is issued with the real path makes this attack impossible.

4 Possible Future Work

While CryptFS offers many security benefits, as outlined above, there are some drawbacks to our implementation that could be addressed in future iterations of the system. The most noticeable system deficiencies concern user experience and can be addressed by improving the following areas:

- **Performance**
  Our system sometimes takes several seconds to respond to basic commands such as changing directories or listing directory contents. For simplicity, in our current implementation, each RPC call is made over a new socket and is RSA-signed. Keeping persistent connections open and using a faster encryption scheme would improve the performance of each RPC call.

- **Uninformative Error Messages**
  The errors shown by CryptFS are the standard errors output by the system and are not user-friendly or descriptive enough to pinpoint a particular problem. Further development on CryptFS would involve more appropriate error messages and would allow a user to list files in a directory that they have the permission to read without displaying misleading errors.

[4]Unfortunately, the fact that file security is enforced via knowing true path names and the fact that security is issued independently per path name (so having read access on a particular directory does not imply read access on every file below that directory) means that renaming directories is not possible in CryptFS. A user issuing a rename would need to send a new tuple for every file underneath the directory, which contains the AES key for that file as well as its new path, and encrypt it with the public RSA key of every user that had access to that file. Since the client does not know all the users who have access to a particular path, and additionally does not necessarily have the AES key for every file underneath that directory, there is no way for the rename to work.

- **Lackluster Tool Set**
  The final major UI improvement to be made in future iterations of CryptFS would be a more consistent and easy-to-use tool set. Currently we have a set of six command line tools that are challenging to use. They could ideally be combined into a single "cryptfs" tool with a more consistent UI.