Project Final Report: Secure Sharing with Satan's File System

Chris Colohan, Chuck Rosenberg, Greg Steffan
Software Systems–December 15, 1997

Abstract

The Secure File System (SFS) is a mechanism which allows the secure sharing of data between machines without direct or secure communication between them. This paper outlines the design of the SFS filesystem, and a prototype implementation. The performance of the system and the security it offers are also analyzed. Finally, future directions and improvements are discussed.

1 Introduction

A brave new venture, Hellfire consulting has just been formed. It has three consultants: Joe Belial who lives in LA, John Beelzebub from Toronto, and Joan Lucifer from Boston. They have just started coding a new software project for a hot new client, and need a way of sharing their code over the Internet while keeping it private. Joe has leased some storage space on a local Internet service provider, but he doesn't trust the ISP to keep their files secure. They want to be sure that only the three of them can read and modify their source code — and so they turned to the Secure File System.

The Secure File System (SFS) is a layer built on top of traditional Unix file systems which offers some guarantees about the origin and accessibility of the files contained on it. It does so making no assumptions about the integrity of the filesystem or administrators of the machine on which it resides. Communication between users is minimized — public keys must be securely exchanged between users once, and after that all communication is done through the file system itself.

The goal of this project is to implement a shared secure file system. Specifically, the implementation ensures that a read of an encrypted file is successful only if that file was written by a trusted entity, and that no entities other than the trusted group members can read the *plain text* contents of that file.

1.1 Design Overview

Since our design is built in a layer on top of existing file systems, all management of the media and all mechanisms for the sharing of files between machines are inherited from these file systems. We assume that the underlying file system is unsecure. Because of this, our implementation does not guard against the deletion or corruption of a file by a super user on a remote file system — however, we will be able to detect tampering with files. In addition, we assume that each local machine is a secure system — i.e., we do not have to protect the local memory of a user's machine and we can assume that information stored there is safe.

The system uses various encryption systems to protect the secure data and meta-data: a private-key based encryption scheme is used to encrypt the data portion of the file, a public-key encryption mechanism is used for key management, and a signature mechanism is used for authentication. We assume that the user's private key and other access related meta-data is stored in a secure and reliable way within the user's local system.

To manage shared access, an access list is associated with each file. The access list allows all enabled users of a file to know the identities of the corresponding trusted readers and writers, and provides them with the key to decrypt its contents. A naive implementation would maintain an access list for each file — this would incur much overhead, and would also require the individual maintenance of each access list. Furthermore, this approach does not mirror the way that shared file systems are typically used: as a means of exchanging data in a group of people working together. Our design overcomes these issues and provides a sharing model which is more useful. We achieve this goal by introducing the notion of a *group*, its *administrator*, and a mapping to the directories where the group is trusted.

A group is a set of users who have read/write access for a specific set of directories. Associated with each group is an access list which contains a list of users in the group and the decryption key for all files owned by the group, which is encrypted under the public key for each authorized user. Each access list is stored as an encrypted file on the shared file system, which we shall refer to as the *group file*. Associated with each group file is a trusted administrator: a special user who is trusted with the maintenance of the group file.

Since each access list will be maintained by the group administrator, a regular user only needs to maintain four pieces of information in his local system to gain access to a shared file: his private key, his public key, the root directory of the sub-tree associated with each group, and the public key of the administrator of each group.

In our implementation, the group file will reside in the root directory for the group, and all sub-directories of the root directory for a group will inherit the group access of the root directory — e.g., `/usr/marketing/reports` inherits access from the root directory `/usr/marketing`. The use of a rooted sub-tree eliminates the need to directly associate each sub-directory or file with a group file.

1.2 Course Material Relationships

This project involves the following material from the course: security and access control; authentication; BAN logic; and file systems.

2 Using SFS

This section presents an overview of how to set up and use our prototype of SFS.

The first thing that is required to use SFS is a shared volume between the machines of the SFS users. This could be a NFS directory, an AFS directory, or any location shared by all of the SFS users. All of the SFS users must be able to read and write this partition. In addition, the full pathnames of files must be the same on each user's machine (which means an NFS partition must have the same mount point).

To install SFS on a user's system, the new `libc.so` must be placed in the dynamic link path (such as in `/lib`), and the executables must be placed in the command search path. Each user of SFS has to have a private directory on their local machine which holds their private key as well as access information for SFS. Once this directory is created, the `SFS_HOME` environment variable must be set to point to it. To generate a public/private key pair, each user must run the `keys_new` utility. This utility generates an RSA key pair using `/dev/random` as a source of a truely random seed. The user's public key can then be printed using the `keys_list` utility, and the public key must then be distributed to the other users of SFS through a reliable channel.

Each user has to have a list of users and their public keys so that SFS can verify the origins of files. This list is stored in `$SFS_HOME/users.sfs`, and is a list of user names and public keys. It is a plain text file, and can be created using a standard text editor. Each user also has a list of SFS file systems that they are participating in, which includes the administrator, full path of the file system, and a version number.

A new file system is created by making a list of authorized users with their desired permissions, and running the `grp_new` utility. Each user has to create a file `$SFS_HOME/groups.sfs` which lists the administrator, location, and version of each secure file system.

Once this is done, the user can treat the secure file system as any other file system. Files that are copied to or edited on the new file system are automatically encrypted before being written to the underlying file system, and any files that are read are automatically decrypted before reading them.

A number of tools are also provided for maintaining the SFS once it is created. `els` will list the files in the current directory including the correct (unencrypted) file size, embedded nonce, and author. `grp_copy` allows the user list and permissions from one file system to be used when creating another one. `revoke` removes a user from the access list of a file system, and re-encrypts all of the files on the file system so that the user can no longer read or write them. After revoking access, the administrator must inform all of the users of the filesystem of the new group file version number to ensure freshness and avoid spoofing attacks.

3 Design

3.1 Data Structures

The secure file system requires the user's private key and the access data for a group in order to access a group file–this information resides in the local, secure system. The access data for a group consists of the absolute pathname of the root directory for that group, and the public key of the administrator of the group. The local data structure for storing access data will have the following format:

$DirectoryTreeName, User_1, K_1$
$DirectoryTreeName, User_2, K_2$
$DirectoryTreeName, User_3, K_3$

The following describes each field: the $DirectoryTreeName$ field maps a directory tree to a specific group, the $User_a$ field is the name of the group administrator and is used for informational purposes, and K_a is the public key for the administrator ($User_a$).

The format for a group file is as follows:

$VersionNumber$
$Filename$
$User_1, \{K_G\}_{K_1}, K_1, WriteAccess_1$
$User_2, \{K_G\}_{K_2}, K_2, WriteAccess_2$
$User_a$
$Signature$

The following describes each field: the $VersionNumber$ field is the current version number of the group file to provide freshness; the $Filename$ field is the absolute pathname of the file; for each user ($User_i$) the group key (K_G) is encrypted under his public key (K_i) and a flag ($WriteAccess_i$) is specified describing his write permission for the group; the field $User_a$ is the name of the administrator; and $Signature$ is a signed digest which verifies the contents of the file.

The format for a regular file is as follows:

$Nonce$
$Filename$
$\{DATA\}_{K_G}$
$User_W$
$Signature$

The following describes each field: the $Nonce$ field is a unique value generated each time the file is written which can be used to provide freshness; the $Filename$ field is the absolute pathname of the file; $\{DATA\}_{K_g}$ is the data portion of the file encrypted under the group key; $User_W$ is the name of the user who last wrote the file; and $Signature$ is a digest, signed by the last writer of the file.

4 Security Evaluation

SFS attempts to provide some security where none existed before using a medium that can store data, but offers no assurance of the integrity or security of the data. On this medium, a hostile attacker can:

- read any files they like;
- alter files as they please;
- erase files;
- replace new versions of files with old versions.

A BAN analysis provided in the appendix outlines the logic that leads us to these beliefs.

The principal design goal of SFS is to provide some security while eliminating the need for secure and timely communication between the users of the file system. Secure and timely communication could be achieved either through a secure messaging protocol between users or through a centralized trusted server — both of which would reduce the usability of the system if they were required. As a result of this, absolute security has not been attained, and the system is still open to spoofing attacks due to the lack of freshness in files.

Since there is no communication between users, a reader of a file has no way of knowing if what is being read is the most recent version of the file. This means that a hostile attacker could substitute older versions of files for newer version without being detected by anyone other than the original writer. To solve this problem, a nonce is inserted in the SFS file which is changed to a unique value every time the file is updated. If a writer of a file communicated the nonce to a reader through an outside secure channel, then the reader could be assured that the file is fresh.

The same problem affects the group files that list who is permitted to access a file system. A hostile attacker could replace a group list file with an older version at any time. As a result, it is impossible to revoke someone's access to a file system — they could easily replace the new group file with the old one, meaning that users other than the administrator would continue to unknowingly share files with that user. Hopefully it is uncommon for a user's access to be revoked, so explicit communication with the SFS users can be used to notify everyone of the revocation. To revoke someone's access, the entire file system is re-encrypted with new keys, and a new group file is created that excludes the revoked user. A version number is attached to the group file, and the version is incremented every time a user is removed from the group. To prevent the spoofing, the administrator has to notify all of the users of the file system of the new group version — once a user is using the new group version they no longer accept files with an earlier version number.

In addition to freshness attacks on SFS itself, there are a number of places where our prototype of SFS has potential holes. Our prototype uses the IDEA algorithm for encrypting file contents, MD5 for computing digests of files, and RSA for key distribution and signing digests. Any attacks on any of those algorithms would also work on SFS.

Our prototype uses a single IDEA key to encrypt all of the files in a file system. This means that there is a lot of data for a cryptanalysis attack on a single key. This could be prevented by limiting the usage of a single key, and having multiple keys per file system.

The regularity of the encrypted files can also be useful for cryptanalysis. Our prototype encrypts file data as written, while compressing the files before encrypting them would make the keys harder to crack as well as conserving storage.

Currently, our prototype allows a symbolic link between a directory on a secure file system and an insecure file system. This could be used to spoof secure files and replace them with a directory of insecure ones. We regard this as a bug in our prototype, but it also can make the secure file system more usable.

5 Performance Evaluation

Our hypothesis as stated in the design document is the following: we can build a secure file system which allows sharing under encryption, but sacrifices performance to do so. The following performance evaluation will show that we have proven the hypothesis with performance degradation to spare. We will investigate the overheads of the SFS system, as well as the performance on realistic benchmarks.

All of our timing measurements were performed on the local system hard disk. The local disk system was chosen over a remote filesystem like AFS to remove the variability in timing that would result from unpredictable AFS loads. Application timings were performed with command line timing. Function call timings were performed with specially written test software which utilized the system interval timer. In all cases, the measurement result reported is the result of averaging at least five measurements together. In the case of the function call timing, error bars are also reported which are +/- a standard deviation.

The overhead of the SFS system includes the storage of metadata in each encrypted file, performing encryption and decryption, using twice the memory to store a file (to encrypt to and from). As shown in Figure 1, the size of the metadata overhead is constant, with a size of roughly 170 bytes depending on the length of the absolute path name.

When an encrypted file is opened under SFS, the entire file is read into memory and decrypted. This adds significant overhead to the opening of files, as shown in Figure 2. For unencrypted files, SFS is about 10 times slower than with SFS disabled due to having to read configuration files. For large encrypted files, the slowdown is significant since decryption is fairly computationally intensive. Figure 3 shows that

Table 1: Slowdown of SFS on benchmarks.

Benchmark	**Slowdown**
co	38.7
make	7.19
latex	6.85

Table 2: Slowdown of SFS on `co` for varying number of users.

Number of Users	**Slowdown**
1	38.7
10	40.2
100	44.3

opening then closing the file performs similarly. These two experiments indicate that SFS will perform poorly for applications which open and close files without doing any work on them.

Since the entire encrypted file is read into memory on open, reading the file should be significantly faster. Figure 4 shows the performance of SFS when files are opened and then read to completion. For large files, the overhead of reading configuration files is almost completely hidden. However, for large encrypted files the overhead of decrypting the file is still dominant.

When writing to an encrypted file, SFS buffers the entire file in memory. When the file is closed, SFS performs encryption and writes the file out with one system call. As shown in Figure 5, writing a large encrypted file in blocks under SFS can outperform an writing to an unencrypted file, since only one system call is made to write the file. Normally, many system calls would be made on each block.

The throughput of SFS is given in Figure 6. Here SFS on encrypted files performs from about 20 to 100 times worse than with SFS disabled, depending on file size.

To measure the performance of SFS under simple usage conditions, we copied files to, from and within an encrypted directory. Figure 7 shows the slowdown of SFS relative to SFS disabled. When copying without encryption, the slowdown of SFS varies from 1 to 5 times, with the exception of a perturbation for the 100 byte file. When copying to or from an encrypted directory slowdown varies from 5 to 25 times, and when copying within an encrypted directory slowdown varies from 5 to 40 times. All copies involving encryption have a peak slowdown for 10KB to 100KB files, since encryption dominates for these file sizes. For files greater than 100KB the slowdown improves due to the fewer numbers of system calls for writing. For large encrypted files the performance degrades again since SFS requires twice the memory for the file (to encrypt/decrypt), and thus the memory requirement approaches the cache size (64MB).

We now investigate the performance of SFS on several realistic workloads: `co`—an RCS check out of 41 files (the source for SFS); `make`— compiling SFS with gcc; and `latex`—compiling the SFS design document. As shown in Table 1, `co` achieves a slowdown of nearly 40 times when checking-out into an encrypted directory. This slowdown is large because `co` does little work on the files, it simply reads and writes them. `make` and `latex` have similar slowdowns of around 7 times when compiling in an encrypted directory.

Finally, we investigate the performance of SFS when varying the number of sharing users in a group. As shown in Table 2, slowdown for `co` increases slightly when the number of users varies from 1 to 100. This is due to the overhead of reading a large access list for every file read.

6 Prototype Implementation

The prototype which we have prepared of SFS under Linux demonstrates the feasibility of the security design. The prototype was created by altering libc.so, so that dynamically linked executables on the system would have their file related system calls intercepted and redirected through the SFS code. In particular, whenever a file is open()ed, it is checked to see if it is on SFS. If it is not, it is passed on to the OS, and all future calls on that file descriptor are also passed to the OS to handle. If the file is on SFS, then the file is read and decrypted into a memory buffer, and all future calls are redirected to the memory buffer. When the file is closed, then it is encrypted and written back to the underlying file system. Ideally, the encryption and decryption should be transparent to applications. So the SFS effectively operates as a layer on top of any existing file system, allowing it to be used to enhance the security of a wide variety of configurations.

Due to time constraints, not every file related system call was fully implemented in SFS, but a subset robust enough for a significant set of Unix tools to work was completed. Of the tools we tested, the ones that work without errors are `cp`, `mv`, `rm`, `emacs`, `netscape`, `rcs`, `gcc`, `make`, `ld`, `ftp`, `latex`, `gv`, `dvips`, `xdvi`, and `xfig`.

What is more interesting is the commands that don't work. These failures can be explained in terms of the system calls we have not implemented in libc. We do not support memory mapped files, so any program that uses mmap() will not work correctly. This includes running executables from our file system, and loading files in StarOffice 4.0 Beta.

The functions `stat()` and `lstat()` return the size of the actual file on disk, including the encryption metadata. This means that commands such as `ls` will report files to be larger than they are if copied to a non-encrypted file system. As a result of this discrepancy, `zip` and `tar` get confused and do not work correctly.

The two problems listed above can be resolved within libc. The third one is not so simple. If a program does a `fork()` followed by an `exec()` (such as a shell loading a program to run it), then the child process is supposed to inherit the parents file descriptors. Our implementation keeps a file descriptor table in memory to track encrypted file information. Since this table is in user memory, it is re-initialized to zero when the new program starts. This means that our library can "forget" that one of the standard streams has been redirected to an encrypted file, since that is determined when the file is opened. As a result, redirecting stdin, stdout, and stderr to or from encrypted files is unreliable, since doing so sometimes bypasses our encryption mechanism producing unencrypted files which our library views as corrupted.

7 Future Work

There is a lot of room for improvement in the design of SFS itself. The SFS design has two weak points, both of which involve freshness in files. Since a writer of a file can only communicate with the reader of the file through the file itself, there is no way of a reader validating the freshness of a file without communicating with the writer. This also applies when removing a user from an access group — there is no way of knowing that an access group is fresh without contacting the group administrator. A more efficient way of communicating freshness information for files and groups needs to be found.

Our prototype of SFS has a lot of room for improvement. Since security is its primary concern of SFS, the security is the first thing that should be addressed. Stronger encryption algorithms and longer keys could be used to improve security. More than one IDEA key could be used for an access group to improve resistance to cryptanalysis. File compression would also strengthen security. When looking at the PGP source code, it tries to erase all traces of unencrypted data from buffers in RAM after they are no longer needed — this practice should be adopted in our prototype. Internal buffers could also be stored in non-pageable RAM, so that unencrypted data is never recorded on a non-volatile medium. Existing, well tested mechanisms for public key storage and distribution could be used, such as using PGP key management and key rings. Easier to use administrative tools would make mistakes less common, and therefore improve security. And formal verification of code that handles unencrypted data and keys could be used to help ensure that they implement the specification of SFS.

Once security is addressed, the usability of SFS is a concern. If SFS is easier to use, then it is more likely to be used. SFS is no protection at all if it is not used. Firstly, SFS should be complete, and offer all the features that other Unix file systems offer. Currently, the inability to mmap() files and the inability to execute files from SFS are the two main holes. These features should be added to make SFS a complete filesystem.

Performance is also important if SFS is to be usable. There are a number of ways that the performance of SFS can be improved. Currently, files are treated as single units, which are encrypted and decrypted all at once. A blocking implementation would be able to address encrypted files a block at a time, and reduce both open and close latency, as well as disk I/O. A blocking implementation would also work around the current problem of the entire file being buffered unencrypted in memory, and hence reduce the memory usage of SFS. If we find that files are not always read in their entirety, a blocking implementation would also help performance by only decrypting the parts of a file that are actually used.

Reliability is also a major concern in any file system. Right now SFS still has a number of unanticipated features which need to be corrected before it would be called robust. Since it is promising security, using program verification to validate that the implementation matches the specification. Error handling could be improved to provide more friendly and informative information to end users.

8 Conclusions

The secure file system was supposed to provide the ability to securely share files on an unsecure medium. We have achieved that in our prototype, at the cost of performance. Our testing was comparing the performance of local disk accesses to encrypted local disk accesses, which is a worst case scenario — when accessing a file remotely over a wide area network, the overhead of our encryption should not be as bad. The prototype has proven the feasibility and usefulness of SFS, and it shows that there is a lot of room for future improvements.

9 Appendix A — Exact File Formats

This appendix describes the exact file formats, byte for byte, that will be used to implement the secure file system. Because it was necessary to agree on some convention for word order, we have decided that all word and long word values will be stored in little endian order.

First, we define some notation:

Symbol	Interpretation
L_{K_G}	the length, in bytes, of a group key, typically 16 bytes
L_{K_U}	the length, in bytes, of a user's public key
L_U	the length, in bytes, of a user's user name including the null
L_S	the length, in bytes, of a signature, typically 16 bytes

The following is the format of a group file:

Byte Offset	Description
$0 - 3$	magic number to identify file
$4 - 5$	version number of the secure file system
$6 - 9$	version number of the group file
$10 - 11$	length of absolute filename field (m) in bytes (includes terminating null)
$12 - 15$	length of data portion of the file in bytes (n)
$16 - (m + 15)$	full absolute path filename, null terminated
$(m + 16) - (m + n + 15)$	data portion of the file - the group access list
$(m + n + 16) - (m + n + L_U + 15)$	user name, null terminated
$(m + n + 16 + L_U) - (m + n + L_U + L_S + 15)$	writer's signature, signature includes all bytes $0 - (m + n + L_U)$ in digest calculation

The group file data is a list of fixed size user entries. The entire list is sorted in ascending order by user name, to facilitate binary search of the data. Each of these entries has the following structure:

Byte Offset	Description
$0 - 31$	user name, null terminated, all unused bytes set to null
$32 - 33$	user's permission flags for the group
$34 - (33 + L_{K_G})$	the group key encrypted with the user's public key
$(34 + L_{K_G}) - (33 + L_{K_G} + L_{K_U})$	the user's public key

The following is the format of a regular file:

Byte Offset	Description
$0-3$	magic number to identify file
$4-5$	version number of the secure file system
$6-9$	nonce for this file
$10-11$	length of absolute filename field (m) in bytes (includes terminating null)
$12-15$	length of data portion of the file in bytes (n)
$16-(m+15)$	full absolute path filename, null terminated
$(m+16)-(m+n+15)$	encrypted data portion of the file
$(m+n+16)-(m+n+L_U+15)$	user name, null terminated
$(m+n+16+Lu)-(m+n+L_U+L_S+15)$	writer's signature, signature includes all bytes $0-(m+n+15)$ in digest calculation

10 Appendix B — BAN Analysis of Scheme

In this section we apply BAN authentication logic [1] to our scheme.

10.1 Additional Notation

First, we augment BAN logic with some additional notation to allow the representation of our scheme.

10.1.1 Sets

We define four sets which enumerate elements of the scheme:

- $USERS$: the enumeration of all users.
- $ADMIN$: the enumeration of all administrators of groups, $ADMIN \subseteq USERS$.
- $ACCESS_G$: the enumeration of all users who have access in a given group G,$ACCESS \subseteq USERS$.

- $GROUPS$: the enumeration of all groups.

10.1.2 Elements

Now we define the following elements:

- U_i: a user where $i \in USERS$.
- K_i: public key for U_i.
- K_i^{-1}: private key for U_i.
- G_j: a group where $j \in GROUPS$.
- K_{G_j}: group key for group G_j.
- D: a digest (digests are taken on the entire file in question).
- F: a file.
- FN: the global, unique path name of a file F.
- DIR: the global, unique path name of a directory.
- GF_j: a group file for group G_j.
- GFN: the global, unique path name of a group file GF_j.
- WA_i: write access for U_i (a boolean value)
- U_a: administrator of a group where $a \in ADMIN$.
- U_m: last modifier of a file where $m \in ACCESS$ and $WA_m = true$.

10.1.3 Operators

We define the following operators:

- $K_i \rightarrow U_i$: U_i has key K_i.
- $WA_i \rightarrow U_i$: U_i has write access WA_i.
- $GF_x \rightarrow DIR_i$: GF_x applies to directory DIR_x.
- $FN\ in\ DIR$: the file name is in the directory.
- $F\ contains\ X$: file F contains element X.
- $D\ certifies\ F$: D is the correct digest of file F.

10.1.4 File Formats

We define the format for group files and regular files:

- $GF\ contains\ (GFN, \{U_i, K_i, \{K_G\}_{K_i}, WA_i | i \in ACCESS\}, \{D\}_{K_a^{-1}})$
- $F\ contains\ (FN, \{Data\}_{K_G}, \{D\}_{K_m^{-1}})$

10.2 BAN Analysis

We now proceed with the BAN analysis of our scheme. For this analysis, we will assume the identity of U_0, an arbitrary user.

10.2.1 Initial Information

We begin the analysis by listing the beliefs which are based on information which is stored in the local system, and is thus trustworthy.

$$U_0\ believes\ K_0 \rightarrow U_0. \tag{1}$$

$$U_0\ believes\ K_0^{-1} \rightarrow U_0. \tag{2}$$

Equation 1 and Equation 2 state our trust in our private and public keys.

$$U_0\ believes\ K_a \rightarrow U_a, \forall a \in ADMIN. \tag{3}$$

So we also have the public key of each group administrator.

$$U_0\ believes\ (G_j\ controls\ DIR), \forall j \in GROUPS. \tag{4}$$

This equation represents the mapping from each group to the directory which it controls. In reality, a group may control more than one directory, but we will continue under this simplified model.

$$U_0\ believes\ (U_a\ controls\ G_j), \tag{5}$$

such that for each $j \in GROUPS$, there exists exactly one $a \in ADMIN$–so each group is controlled by exactly one administrator.

$$U_0\ believes\ (U_a\ controls\ K_{G_j}), \tag{6}$$

such that for each $j \in GROUPS$, there exists exactly one $a \in ADMIN$–this states our belief that the administrator also controls the key for the group.

$$\frac{U_0\ believes\ (U_a\ controls\ G), U_0\ believes\ (G\ controls\ DIR)}{U_0\ believes\ (U_a\ controls\ DIR)}. \tag{7}$$

So by associativity of control, we believe that the group administrator may be trusted in files located in the directory.

10.3 Authentication of a Group File

Now we will analyze the authentication of a group file. To reduce the amount of notation in this proof, we will omit the indices from group files, groups, and directories since we are dealing with exactly one of each.

For group file GF, where $GF\ in\ DIR$, we begin by reading the group file:

$$U_0\ sees\ GF. \tag{8}$$

From 3 and 8, we get:

$$\frac{U_0\ believes\ (K_a \rightarrow U_a), GF\ contains\ \{D\}_{K_a^{-1}}}{U_0\ believes\ (U_a\ said\ D)}. \tag{9}$$

Since the digest is signed by the administrator, we believe that the administrator computed the digest. Using 9 and the digest:

$$\frac{U_0\ believes\ (U_a\ said\ D), D\ certifies\ GF}{U_0\ believes\ (U_a\ said\ GF)}. \tag{10}$$

Since the digest coincides with the content of the group file, we also believe that the administrator created the group file. We use our mapping from groups to directories (4), the global file name included in the group file:

$$\frac{U_0\ believes\ (U_a\ said\ GF), GF\ contains\ GFN, GFN\ in\ DIR}{U_0\ believes\ (U_a said (GF\ in\ DIR))}, \tag{11}$$

to assert that the administrator placed this group file in this directory. Using 11 and the fact that the administrator may be trusted in the directory (7):

$$\frac{U_0\ believes\ (U_a\ said\ (GF\ in\ DIR)), U_0\ believes\ (U_a\ controls\ DIR)}{U_0\ believes\ (U_a\ controls\ GF)}, \tag{12}$$

so the group file is controlled by the administrator since it came from the directory where we trust the administrator. Using 12 and the jurisdiction rule, we get:

$$\frac{U_0\ believes\ (U_a\ believes\ GF), U_0\ believes\ (U_a\ controls\ GF)}{U_0\ believes\ GF}. \tag{13}$$

Because we trust the administrator for this group, and that administrator created the group file, we may now trust the group file. Since we believe the group file, we therefore also believe its contents:

$$\frac{U_0\ believes\ GF, GF\ contains\ (U_i, K_i, WA_i)}{U_0\ believes\ (K_i \rightarrow U_i, WA_i \rightarrow U_i)}, \forall i \in ACCESS_G. \tag{14}$$

The access list of users, their public keys, and their write access are now available.

$$\frac{U_0\ believes\ GF, GF\ contains\ \{K_G\}_{K_0}, U_0\ believes\ (K_0 \rightarrow U_0)}{U_0\ believes\ K_G \rightarrow G}. \tag{15}$$

Assuming that we have access to this particular group, we may decrypt the key for the group using our private key, and we may trust that this is the proper group key. Finally, we must verify that this is the proper group file for the directory we are accessing (in case an evil user has switched around group files.

10.4 Authentication of a File

Now that we have authenticated the group file, we may proceed to authenticate the target file itself.

For file F, where $F\ in\ DIR$, we first read the file:

$$U_0\ sees\ F. \tag{16}$$

Since the file contains the global file name, we can look-up the group file that applies using 4 and 12:

$$\frac{F\ contains\ FN, FN\ in\ DIR, U_0\ believes\ G\ controls\ DIR}{U_0\ believes\ GF \rightarrow F}. \tag{17}$$

Using this result, 4 and 18:

$$\frac{U_0 \; believes \; GF \rightarrow F, F \; contains \; \{D\}_{K_m^{-1}}, U_0 \; believes \; K_m \rightarrow U_m}{U_0 \; believes \; (U_m \; said \; D)}. \tag{18}$$

The digest is signed by the user to last modify the file. We use the group file to look-up the public key for the user, and verify that the user computed the digest. Using this fact:

$$\frac{U_0 \; believes \; (U_m \; said \; D), D \; certifies \; F}{U_0 \; believes \; (U_m \; said \; F)}. \tag{19}$$

So the user also created the file, since the digest coincides with the file. Using the group file (14):

$$\frac{U_0 \; believes \; (U_m \; said \; F), U_0 \; believes \; WA_m \rightarrow U_m, WA_m = true}{U_0 \; believes \; F}. \tag{20}$$

This states that if this user has valid write access according to the group file, then we may believe that the contents of the file are valid. Using the group key (15), we may decrypt the data, and trust that it is valid:

$$\frac{U_0 \; believes \; F, F \; contains \; \{Data\}_{K_G}, U_0 \; believes \; K_G \rightarrow G}{U_0 \; believes \; Data}. \tag{21}$$

The following fact is obviated by the BAN logic: since we do not deal with *nonces* or the concept of *freshness*, we do not have protection against attacks which replace newer versions of files with older ones. However, this attack does not violate our original goal to ensure that only authorized users will be able to read the content of shared files. It does mean that we may not support the semantics of deleting a user from a group once the group file has been made public. An evil user who is deleted from the group could add himself back to the group by replacing the new group file with the old one which listed him as a user. In order to support deletion of users from groups, we have added a nonce to each file which must be verified by a secure means outside of SFS.

10.5 Other Actions

It is not necessary to list the ban logic for other types of file accesses, since they do no more than trust the information in the group files, as already shown. For example, to create a new group file, the administrator need only trust his local list of public keys, and then create the group file (in proper form). When any user in a given group creates a new file, the group file is consulted to obtain the group key, and then the file is created. A similar process is performed for updating an existing file.

11 Appendix C — Syscalls Intercepted in `Libc`

The following syscalls are intercepted to implement SFS, and are functional in our prototype:

close Close a file. If the file is encrypted, encrypt the buffered contents, write them out, and sign the file.

creat Create a new file. If encrypted, allocate a buffer to hold the contents of the file.

dup,dup2 Duplicate a file descriptor. If the file is encrypted, the duplication should be tracked.

fcntl Manipulate file descriptor. No special actions are taken for an encrypted file.

fstat Returns status information about the specified file. If the file is encrypted, then the meta-data size has to be subtracted from the returned length.

ftruncate Truncate a file to a specified length. If the file is encrypted, then it should be decrypted, truncated, then re-encrypted.

lseek Reposition read/write file offset. If the file is encrypted, keep a copy of the new offset locally for future reads and writes.

open Open and possibly create a file. If the file is encrypted, decrypt and read the entire contents of the file into a buffer.

read Read bytes from a file descriptor. If it is encrypted, simply read from the buffer.

rename Change the name or location of a file. If the file is encrypted, it needs to be decrypted before the move, and possibly reencrypted after the move.

write Write bytes to a file descriptor. If it is encrypted, simply write to the buffer.

The following syscalls are intercepted to implement SFS, but have not yet been completed in our prototype:

chroot Change root directory. All encrypted filenames should be adjusted to reflect this change.

exit Close any open files. If a file is encrypted, encrypt its buffered contents, write them out, and sign the file.

llseek Reposition read/write file offset. If the file is encrypted, keep a copy of the new offset locally for future reads and writes.

mmap,munmap Map or unmap files or devices into memory.

readv,writev Reads and writes to / from vectors.

stat,lstat Returns status information about the specified file. If the file is encrypted, then the meta-data size has to be subtracted from the returned length.

sync Commit buffer cache to disk. If there are encrypted files open, then they should be first written to disk.

truncate Truncate a file to a specified length. If the file is encrypted, then it should be decrypted, truncated, then re-encrypted.

References

[1] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.