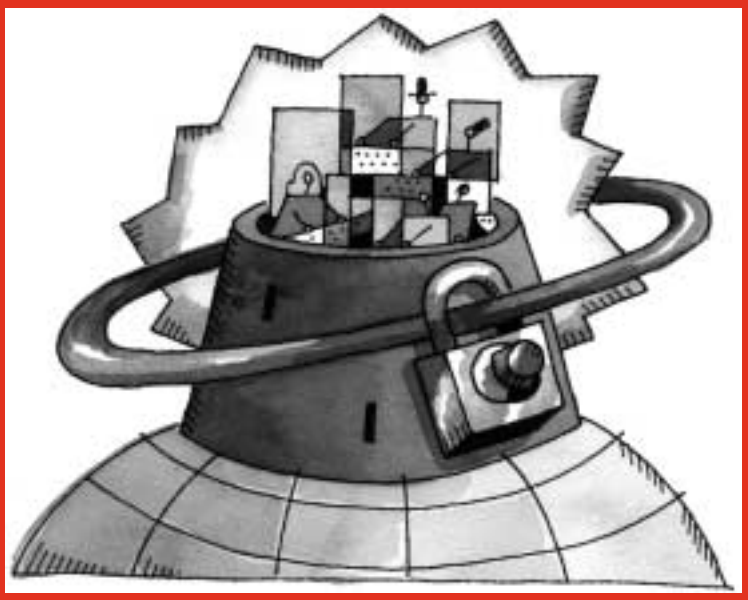
;login:
THE MAGAZINE OF USENIX & SAGE
December 2002 • volume 27 • number 6

Focus Issue: Security

Guest Editor: Rik Farrow

inside:

Fu, Kaminsky, and Mazières: Using SFS for a Secure Network File System



USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

using SFS for a secure network file system

by Kevin Fu,

Kevin Fu is a doctoral student at the Laboratory for Computer Science at MIT. His research interests are computer security, cryptography, and operating systems.

fubob@mit.edu

Michael Kaminsky,

Michael Kaminsky is a doctoral student at the Laboratory for Computer Science at MIT. His research interests are operating systems, security, and networking.

kaminsky@lcs.mit.edu

and David Mazières

David Mazières is an assistant professor of computer science at New York University. He began the SFS project while a doctoral student at MIT.

dm@cs.nyu.edu

Introduction

The Self-Certifying File System (SFS) (*http://www.fs.net/*) is a secure distributed file system, and associated utilities, that can both increase the security of networks and simplify system administration. SFS lets people securely access file systems over insecure networks without the need for virtual private networks (VPNs) or a public key infrastructure (PKI). In many situations, SFS provides a suitable and more secure alternative to the widely deployed NFS file system. Nonetheless, SFS gracefully coexists with NFS and other file systems such as Samba. Thus, one can easily install, test, and incrementally deploy SFS without disrupting existing network services.

SFS administration is greatly simplified by the fact that client machines have no configuration options. An SFS client just needs to run a program called sfscd at boot time; users can then access any server with no further administrative assistance. In contrast, many distributed file systems require the client to have a list of what remote file systems to mount where. In SFS, it is actually the server that determines which of its file systems clients mount on what pathnames. As users access those pathnames, client machines learn of new servers and transparently "automount" them.

SFS cryptographically secures all client-server network communications with encryption and a message authentication code. To prevent rogue servers from impersonating valid ones, each SFS server has a public key. A server's files all reside under a so-called *self-certifying pathname* derived from its public key. Self-certifying pathnames contain enough information for an SFS client to connect to a server and establish a cryptographically secure channel – even if the client has only just learned of the server through a user's file access. A variety of techniques exist for users to obtain servers' self-certifying pathnames securely.

In addition to protecting network traffic, SFS performs user authentication to map remote users to credentials that make sense for the local file system. SFS's user-authentication mechanism is based on public key cryptography. On the client side, an unprivileged *agent* process holds a user's private keys and transparently authenticates him or her to remote servers as needed. On the server side, SFS keeps a database of users' public keys. Users can safely register the same public key in multiple administrative realms, simplifying the task of accessing several realms. Conversely, administrative realms can also safely export their public key databases to each other. A file server can import and even merge user accounts from several different administrative realms.

Background

The SFS project began in 1997 after one of the authors became frustrated by the lack of a global, secure, decentralized network file system. No existing file systems had all three properties.

NFS does not have a viable notion of security for most environments. NFS essentially trusts client machines, allowing an attacker to impersonate a legitimate user with little effort. NFS also lacks a global namespace, because client administrators can mount

NFS file systems anywhere. The same files might be accessible under two different paths on two different machines.

The Andrew File System (AFS) offers a global namespace and, in some contexts, security, but it cannot guarantee data integrity to users who do not have accounts in a server's administrative realm. Moreover, when users have several accounts in different realms, AFS makes it hard to access more than one account at a time. As a result, AFS tends to lead to inconveniently large administrative realms – often as large as an entire campus. Such unwieldy realms in turn restrict users who might want greater autonomy. For example, it is not uncommon for AFS users to have to recycle "guest" accounts to avoid the onerous burden of going through a central administration to create a new account for every visitor. SFS, in contrast, lets a server operator both import a campus-wide user database and manage additional local accounts.

NFS does not have a viable notion of security for most environments.

STATUS

SFS is free software and runs on UNIX operating systems that have solid NFSv3 support. We have run SFS successfully on recent versions of OpenBSD, FreeBSD, MacOS X, OSF/1, Solaris, and several Linux distributions. Although a Windows port exists, it relies on a commercial NFS implementation. We currently have no plans to merge the Windows port into the mainline distribution, though we would like to see this happen eventually.

There are pre-packaged SFS distributions available for Debian, Red Hat, and FreeBSD, among others. The packages greatly simplify installation, because SFS requires a robust compiler that can cope with extensive use of C++ templates. (Some versions of GCC generate internal compiler errors when compiling SFS from scratch.) Because SFS is open source, US regulations allow export of the code to most countries.

The SFS installation and setup procedures are described below. The SFS Web site (*http://www.fs.net/*) has technical papers which provide a detailed discussion of related work, a performance analysis, and the theory behind SFS.

OVERVIEW

From a system administrative point of view, SFS consists of two programs run at boot time. SFS clients must run the SFS client daemon (`sfscd`), which creates the `/sfs` directory and implements the auto-mounting of remote SFS servers. SFS servers must run the SFS server daemon (`sfssd`), which makes local file systems available to SFS clients on TCP port 4.

Internally, however, SFS is structured in a modular manner and consists of many daemons. `sfscd` and `sfssd` each launch and communicate with a number of subsidiary daemons that actually implement the different SFS features (Figure 1). For example, `sfscd` is responsible for automatically mounting new remote file systems. On the server machine, `sfssd` accepts incoming SFS connections and de-multiplexes these requests to the appropriate SFS server daemons. The three basic servers are the SFS file server, the authentication server, and the remote login server. The client and server file system daemons communicate with the kernel using NFS loopback. SFS components communicate with each other using RPC. SFS implements a secure RPC transport which provides confidentiality and integrity. Finally, the SFS agent runs on the client machine, one instance per user.

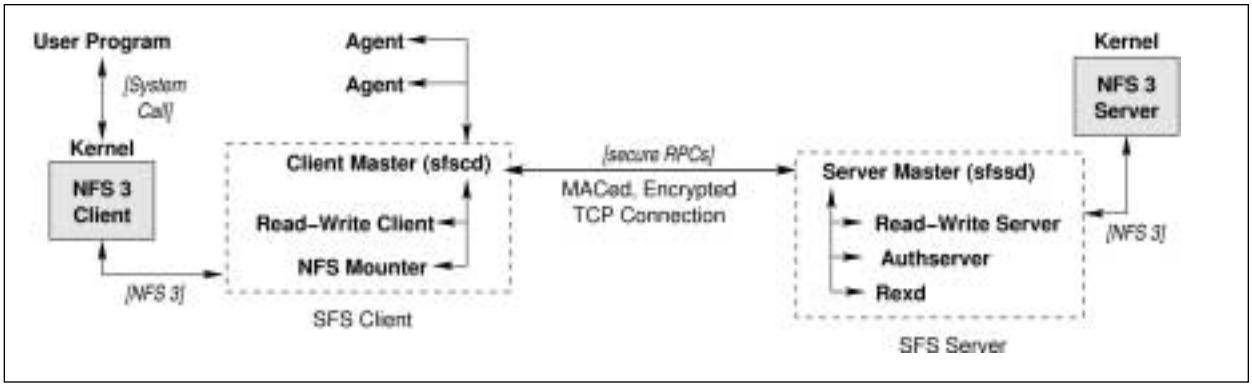


Figure 1: Architectural overview of SFS

If you already use NFS, switching to SFS is straightforward. Because the SFS server can coexist with an NFS server on the same machine, you don't have to switch "cold turkey" from NFS to SFS: you can serve both from the same machine. For instance, you may wish to continue using NFS for local machines but make SFS available for users traveling or using wireless networks.

A single machine can be both an SFS client and a server. However, the client software will refuse to mount an SFS file server running on the same machine, since this can cause deadlock in the kernel on some operating systems. This is a result of SFS's implementation as an NFS loopback server for portability.

Naming Servers

A fundamental problem that SFS tries to address is how to name resources securely. Setting up secure communication requires authentication of the remote resource; systems today often use some form of public key cryptography. The basic problem facing these public key–based systems is key distribution: how does the user who wants to connect securely to a remote resource get that resource's public key securely?

Self-Certifying Pathnames

SFS does not mandate any particular kind of key distribution, but instead provides a flexible set of options based on self-certifying pathnames. Given a self-certifying pathname, the public key of the file server can be certified with no additional information. Thus, if one obtains a self-certifying pathname in a trusted manner, the SFS client will automatically verify the associated server's public key.

Figure 2 shows the two basic components of self-certifying pathnames: the location and HostID. The location is the DNS hostname of the file server; it tells the client software where to find the server, but not how to communicate securely with it. The

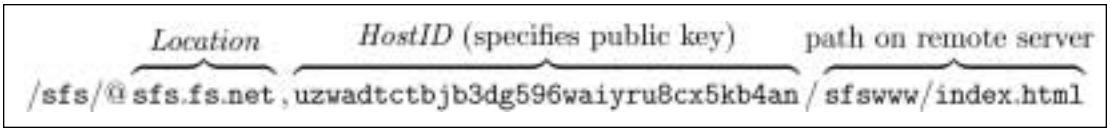


Figure 2: A self-certifying pathname

HostID portion of the self-certifying pathname is what allows secure communication. The HostID is a SHA-1 hash of the server's public key, similar in concept to a PGP fingerprint. Hence, the client can insecurely ask the server for its public key, then verify

that the public key actually matches the hash in the pathname. A user who trusts in the authenticity of the HostID can then trust in the authenticity of the corresponding public key that the client has fetched and verified.

When a user accesses a self-certifying pathname (under /sfs), the SFS client software contacts the server based on the location component. The server responds with its public key. If the server can prove its identity by demonstrating it has the corresponding private key, the SFS client will automatically mount the file server under /sfs. All communication between the client and server is encrypted and mutually authenticated. We will discuss user authentication shortly.

While self-certifying pathnames require some method for users to obtain HostIDs securely, they are not wedded to any particular public key infrastructure or method of key distribution. The user is free to decide, even on a host-by-host basis, how to obtain the HostID of a server. Users who need a centralized repository of public keys (a certification authority) and those who want to set up stand-alone servers both benefit from self-certifying pathnames. The following sections demonstrate several useful techniques for securely obtaining servers' self-certifying pathnames.

Symbolic links provide an easy way for users to refer to a self-certifying pathname.

Symbolic Links as Simple Certificates

Symbolic links provide an easy way for users to refer to a self-certifying pathname. For example, a user might create a symbolic link as follows:

```
redlab -> /sfs/@redlab.lcs.mit.edu,gnze6vwxtwssr8mc5ibae7mtufhphzsk
```

When the user accesses the path redlab, the SFS client software will mount the file server at *redlab.lcs.mit.edu* with the given public key hash. Symbolic links allow users to assign human-readable names to hard-to-type self-certifying pathnames. The user could store these links in a subdirectory of his or her home directory, and they would serve a similar function to the user's SSH known_hosts file.

Administrators can use symbolic links to provide a system-wide set of names for commonly used file servers. These may be distributed using a floppy disk, rsync/rdist over SSH, or whatever other technique is in use for remotely administering software on machines. Once a single symbolic link has been installed on a client, administrators can bootstrap a larger set of self-certifying pathnames and symbolic links from a trusted source. These links can be placed in a well-known location (e.g., /sfslinks) and would serve a similar function to a host-wide SSH /etc/ssh/ssh_known_hosts file.

Symbolic links in the file system have an advantage over a known_hosts-style name cache, because a name lookup can involve several levels of symbolic links; moreover, these symbolic links can reside on multiple file systems, some of which are on SFS. This feature of symbolic links and self-certifying pathnames provides a convenient, simple way to implement certificate authorities as file servers. As a hypothetical example,

```
/sfs-CAs:
        sun -> /sfs/@sfs.sun.com,ytzh5beann4tiy5aeic8xvjce638k2yd
        thawte -> /sfs/@thawte.com,...
        ...

/sfs/@sfs.sun.com,ytzh5beann4tiy5aeic8xvjce638k2yd:
        yahoo -> /sfs/@www.yahoo.com,...
        redhat -> /sfs/@redhat.com,...
        ...
```

Certification programs provide the user with a lot of flexibility.

If a user wants to read from the file server which certificate authority Sun calls redhat, the user can reference a path such as:

```
cat /sfs-CAs/sun/redhat/README
```

The directory /sfs-CAs is on the local machine, the directory /sfs-CAs/sun is on the machine sfs.sun.com, and the directory /sfs-CAs/sun/redhat is on the machine redhat.com (as is the file README). The pathname by which one accesses a file determines how the file server is authenticated. For example, the same README file might be available as:

```
/sfs-CAs/thawte/redhat-server/README
```

Server Nicknames

Some users might require a more sophisticated means of naming a server. SFS allows users to invoke arbitrary certification programs to map a human-readable "nickname" into a self-certifying pathname. The user's agent keeps a list of these certification programs and runs them as needed.

Server nicknames are non-self-certifying names that a user accesses under /sfs (e.g., /sfs/work or /sfs/lab-machine1). When the SFS client software sees a nickname, it asks the user's agent to translate the nickname into a self-certifying pathname. The agent will invoke, in order, all of the certification programs that the user has registered with it until there is a successful match. The sfskey utility, discussed in the examples, allows users to register certification programs with their agents.

Certification programs provide the user with a lot of flexibility. For example, the certification program might look up the nickname in an LDAP database. As a less complex example, the SFS distribution actually comes with a program called dirsearch that takes a list of directories and looks up the nickname in each one until it finds a symbolic link with that name. For instance, the certification program

```
dirsearch ~/my-links /sfslinks/sfs.mit.edu /sfs-CAs/sun
```

would have the effect of giving precedence to the user's personal links directory first, then some university-wide directory, and finally Sun's. The dirsearch program allows the user to specify his or her own trust policy.

Secure Remote Password Protocol (SRP)

SFS also provides a means of securely downloading a server's self-certifying pathname with a password. In this case, the password typed by the user is actually used to authenticate the server to the user in addition to the more conventional authentication of user to server. Though users cannot be expected to remember self-certifying pathnames, they will remember passwords of their own choosing. In this way, users can always resort to typing their passwords if there is not a more convenient way of obtaining a server's pathname.

SFS uses the Secure Remote Password Protocol (SRP) for password authentication.[1] SRP allows users both to download self-certifying pathnames securely and to download encrypted copies of their own private keys. When a user registers a public key with an SFS server, the user can additionally give the server an encrypted copy of the corresponding private key and a secret "SRP parameter" computed as a function of his or her password and the server's location. When the user and server later engage in the SRP protocol, SRP mutually authenticates the two sides' communications. The details

1. Thomas Wu, "The Secure Remote Password Protocol," *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, San Diego, CA, March 1998, pp. 97-111.

of SRP are handled for the user by the sfskey program (or indirectly through sfsagent, which can invoke sfskey). After a successful run of SRP, sfskey installs a symbolic link from the server's location to its self-certifying pathname. For example:

```
/sfs/redlab.lcs.mit.edu ->
/sfs/@redlab.lcs.mit.edu,gnze6vwxtwssr8mc5ibae7mtufhphzsk
```

We chose SRP rather than a simpler protocol so as to protect weak passwords against offline "dictionary attacks." In other file systems, such as AFS, an attacker can exchange a single message with the server, then invest a large amount of computation to guess and verify an unbounded number of candidate passwords. SRP allows only "online" password-guessing attacks – the number of passwords an attacker can guess is proportional to the number of messages the attacker has exchanged with the server or user.

Examples

Many operating system distributions offer pre-compiled SFS binary packages. These packages provide both client and server support out-of-the-box. Other users will most likely have to compile and install SFS from the source. The SFS Web site contains details about the compilation process and about compiler compatibility. Because the client software is implemented as an NFS loopback server, all SFS installations require working NFSv3 client support.

Installing the SFS Client

Installing the SFS client is straightforward using package management tools such as dpkg or RPM. If you are behind a firewall, you will need to allow outgoing connections to TCP port 4. The following example shows how to set up SFS on a freshly installed Red Hat 7.3 box:

```
[root@client /root]# rpm -ivh sfs-0.7-1.i386.rpm
Preparing...  ########################################### [100%]
   1:sfs      ########################################### [100%]
[root@client /root]# /etc/rc.d/init.d/sfscd start
Starting sfscd:                                    [  OK  ]
```

SFS clients require no configuration. Simply run the program sfscd as shown above, and a directory /sfs should appear on your system. To test your client, access our SFS test server. Here we download a file:

```
$ cd /sfs/@sfs.fs.net,uzwadtctbjb3dg596waiyru8cx5kb4an
$ cat CONGRATULATIONS
You have set up a working SFS client.
```

Note that the /sfs/@sfs.fs.net:... directory does not need to exist before you run the cd command. SFS transparently mounts new servers as you access them.

User Authentication

In the example above, the SFS server is exporting its file system publicly and read-only. Typically, however, SFS servers will require user authentication, so that only registered users can access the file system. A registered user is one whose public key has been installed on the SFS server: specifically, in the "authserver." To register a public key, log into the file server and run the command:

```
$ sfskey register
```

Many operating system distributions offer pre-compiled SFS binary packages.

SRP allows the user to securely download a copy of the server's self-certifying hostname.

This will create a public-private key pair for you and register the public key with the server. Note that if you already have a public key on another server, you can reuse that public key by giving sfskey your identifier at that server (i.e., sfskey register user@other.server.com).

The SFS user registration process also sets up SRP. As mentioned above, SRP allows the user to securely download a copy of the server's self-certifying hostname. SFS also uses SRP to store an encrypted copy of the user's private key on the server. The user can then download a copy of his or her private key from the server knowing only a password. Because the private key is encrypted, the server does not have access to it.

In some settings, users do not have permission to log into file servers and thus cannot run the sfskey register command. In this case, there are two options for creating user accounts. The system administrator can ask the user to supply a public key, or else can ask the user for an initial password and use the password to register a temporary public key and SRP parameter.

Running the SFS User Agent

Once you have registered your public key with an SFS server, you must run the sfsagent program on an SFS client when you wish to access the server. On the client, run the command:

```
$ sfsagent user@my.server.com
Passphrase for user@my.server.com/1024:
```

my.server.com is the name of the server on which you registered. user is your identifier on that server. (The value 1024 is the size in bits of SRP's cryptographic parameters, which you can ignore, though paranoid users may wish to avoid small values.) This command does three things: it runs the sfsagent program, which persists in the background to authenticate you to file servers as needed; it fetches your private key from the server and decrypts it using your password and SRP; and, finally, it fetches the server's public key and creates a symbolic link from /sfs/my.server.com to /sfs/@my.server.com,HostID. Each user has a different view of the /sfs directory. Thus, one user's links in /sfs will not be visible to another user, and two users' links will not interfere with each other.

If, after your agent is already running, you wish to fetch a private key from another server or download another server's public key, you can run the command:

```
$ sfskey add user@myother.server.com
Passphrase for user@myother.server.com/1024:
```

In fact, sfsagent runs this exact command for you when you initially start it up. Note that sfskey does not take a self-certifying pathname as an argument; the user's password and SRP are sufficient to authenticate the server holding your encrypted private key.

To generate a public-private key pair explicitly and save it to disk, use the following command:

```
$ sfskey gen
```

Optional arguments allow you to specify the key size and name. The sfskey subcommands edit, list, delete, and deleteall manage the keys stored in your SFS agent. The

sfskey update command allows you to replace the key stored on a server with a new one.

When you are done using SFS, you should run the command

```
$ sfskey kill
```

before logging out. This will kill your sfsagent process running in the background and get rid of the private keys it was holding for you in memory. There is also an option to specify a timeout to automatically remove keys from memory.

Setting Up an SFS Server

Setting up an SFS server requires very little configuration. The procedure consists of setting up the NFS loopback and choosing what directory trees to export. For extra security, you may wish to configure local firewall rules to prevent non-local users from probing portmap. Recall that only the server itself needs to access the NFS exports. Here we start with a Red Hat 7.3 box that has the SFS client software already installed.

```
[root@server /root]# rpm -ivh sfs-servers-0.7-1.i386.rpm
Preparing...          ########################################### [100%]
      1:sfs-servers   ########################################### [100%]
[root@server /root]#
```

Let's assume you want to export disks mounted on /disk/disk0 and /disk/disk1 to authorized users. We first need to create mount points for the SFS root file system and for each exported directory tree.

```
[root@server /root]# mkdir -p /var/sfs/root/disk0 /var/sfs/root/disk1
```

Recall that for portability reasons, SFS accesses the file systems it exports by pretending to be an NFS client over the loopback interface. Thus, the server must export the desired file systems to the localhost via NFS. We add the following three lines to the /etc/exports file to enable NFS exporting of /var/sfs/root, /disk/disk0, and /disk/disk1 to the localhost:

```
[root@server /etc]# cat  /etc/exports
/var/sfs/root localhost(rw)
/disk/disk0 localhost(rw)
/disk/disk1 localhost(rw)
```

Now we start the NFS server:

```
[root@server /etc]# /etc/rc.d/init.d/portmap status
portmap (pid 643) is running...
[root@server /etc]# /etc/rc.d/init.d/nfs start
Starting NFS services:                     [  OK  ]
Starting NFS quotas:                       [  OK  ]
Starting NFS mountd:                       [  OK  ]
Starting NFS daemon:                       [  OK  ]
```

We're almost done! Now we need to create the server's public-private key pair and tell the SFS server to also export the same directories:

```
[root@server /root]# sfskey gen -P /etc/sfs/sfs_host_key
Creating new key for /etc/sfs/sfs_host_key.
      Key Name: root@my.server.com

sfskey needs secret bits with which to seed the random number generator.
Please type some random or unguessable text until you hear a beep:
```

Setting up an SFS server requires very little configuration.

```
DONE
[root@server /root]# cat  /etc/sfs/sfsrwsd_config
Export /var/sfs/root / R
Export /disk/disk0 /disk0
Export /disk/disk1 /disk1
```

The R flag makes /var/sfs/root globally readable – you can omit this flag if you do not wish to have any anonymously accessible directories. Finally, we start the server:

```
[root@server sfs]# /etc/rc.d/init.d/sfssd start
Starting sfssd:                                            [  OK  ]
[root@server sfs]# tail /var/log/messages
Sep 12 23:46:43 server sfssd: sfssd startup succeeded
Sep 12 23:46:43 server : sfssd: version 0.7, pid 1881
Sep 12 23:46:43 server : sfsauthd: version 0.7, pid 1885
Sep 12 23:46:44 server : sfsauthd: serving
@server.mit.edu,66zrwhw5i9jr5ym7i9mkcxijn5fmtaz8
Sep 12 23:46:44 server : sfssd: accepted connection from 18.247.7.168
Sep 12 23:46:44 server rpc.mountd: authenticated mount request
        from localhost.localdomain:790 for /disk/disk0 (/disk/disk0)
Sep 12 23:46:44 server rpc.mountd: authenticated mount request
        from localhost.localdomain:790 for /disk/disk1 (/disk/disk1)
Sep 12 23:46:44 server rpc.mountd: authenticated mount request
        from localhost.localdomain:790 for /var/sfs/root (/var/sfs/root)
Sep 12 23:46:44 server : sfsrwsd: version 0.7, pid 1886
Sep 12 23:46:44 server : sfsrwsd: serving /sfs/@server.mit.edu,
        66zrwhw5i9jr5ym7i9mkcxijn5fmtaz8
```

Remote Login

REX is an SSH-like remote login tool that uses self-certifying paths instead of a static known_hosts file. While REX can be disabled on the server by commenting out a single line in the sfssd_config configuration file, it makes remote login more pleasing to users with home directories stored in SFS. By default, REX forwards X11 connections and forwards the SFS agent itself. The basic invocation is with a hostname:

```
$ rex amsterdam.lcs.mit.edu
rex: Prepending '@' to destination 'amsterdam.lcs.mit.edu' and attempting SRP
Passphrase for fubob@amsterdam.lcs.mit.edu/1024:
rex: Connecting to @amsterdam.lcs.mit.edu,bkfce6jdbmdbzfbct36qgvmpfwzs8exu
amsterdam:(~/)%
```

REX can use the SFS agent and/or SRP to map DNS hostnames to self-certifying pathnames as described earlier. In the above example, REX prompts for a password and uses SRP to obtain amsterdam's self-certifying pathname and the user's private key securely. Subsequent logins to the same server do not require a password.

REX also accepts other names for servers. For example, REX accepts self-certifying pathnames and even SFS symbolic links as a way of naming servers. System administrators may find fully qualified self-certifying pathnames useful for non-interactive scripts. Here we generate a list of currently logged-in users:

```
$ rex @amsterdam.lcs.mit.edu,bkfce6jdbmdbzfbct36qgvmpfwzs8exu
/usr/bin/who
```

Connection caching allows subsequent REX executions to the same server to avoid public key operations. The client and server generate a new session key based on the

previous one. The speedy reconnection will be useful for system administrators who frequently make multiple remote execute commands to the same servers. No longer will each command require a good portion of a second to complete.

You can see what sessions your agent currently maintains by running

```
$ sfskey sesslist
```

The additional command sfskey sesskill removes a connection from the agent's cache.

SSH was the main inspiration for REX, as we needed an SSH-like tool that could work with SFS. Although we could have extended SSH for this purpose, SSH servers typically read files in users' home directories during user authentication. This policy is incompatible with our goal of integrating remote login with a secure file system, as the remote login machine would generally not have permission to read users' files before those users are authenticated.

For those who are hesitant to use REX but need remote login to work with home directories stored in SFS, the libpam-sfs module may be a reasonable alternative.

SFS Toolkit

The SFS file system infrastructure or "toolkit" provides support for several other projects and extensions. Below is a short list of what the extended SFS world has to offer. For all the details and references, see the SFS Web page.

File Systems

One of the goals of SFS is to facilitate the less painful development of new file systems. As an example, one of the authors wrote a crude encrypted file system in just 600 additional lines of C++ code. SFS provides an asynchronous library and an efficient NFS loopback server. By implementing the server side of the NFS loopback server, a developer can create a new file system that works on all operating systems with solid NFSv3 support.

In a read-only dialect of SFS (TOCS 20(1)), a publisher could replicate content on untrusted servers while maintaining the integrity of the file system as a whole. The read-only dialect is significantly faster than the read-write dialect, because the server performs no online cryptography. Another dialect caters to low-bandwidth connections (SOSP 2001). The Chord system (SIGCOMM 2001) uses the SFS asynchronous library to implement a peer-to-peer lookup algorithm. The Cooperative File System (SOSP 2001) uses the SFS asynchronous library to implement a distributed peer-to-peer read-only file system. Ivy (OSDI 2002) does the same for a read-write log-based file system.

ACLs

One drawback to traditional UNIX file sharing is the lack of access control lists (ACLs). We find that file sharing in NFS (and hence SFS) is limited because of the difficulty of creating and administering groups for users without administrative privileges. In a dialect of SFS under development, the server supports ACLs similar to those of AFS. A key difference is that we have no-hassle cross-realm support. A student at MIT can give a friend at NYU access to a file or directory simply by adding the friend's public key to the appropriate ACLs. Although the ACL prototype works and appears in the main sourcetree, it does not yet appear in an official release.

One of the goals of SFS is to facilitate the less painful development of new file systems.

. . . we have never lost a file to SFS.

Caveat

SFS serves files to clients using cryptographically protected communications. As such, SFS can help eliminate security holes associated with insecure network file systems and let users share files where they could not do so before. That said, we realize that perfect security is a remote fantasy. Our documentation on the SFS Web site discusses many of the security issues raised by SFS.

Conclusion

Our research groups have used SFS on a daily basis for several years. Several of us use SFS for our home directories. In part because SFS is implemented on top of mature NFS code, we have never lost a file to SFS. A number of groups outside of our research groups also use SFS in production environments. We hope you find SFS as convenient and useful as we have.

ACKNOWLEDGMENTS

Several people have contributed to the SFS development, including Chuck Blake, Benjie Chen, Frank Dabek, David Euresti, Kevin Fu, Frans Kaashoek, Michael Kaminsky, Maxwell Krohn, David Mazières, Eric Peterson, George Savvides, and Nickolai Zeldovich. We thank Eric Anderson, Sameer Ajmani, Michael Barrow, Rik Farrow, Maxwell Krohn, Chris Laas, Emil Sit, and Ram Swaminathan for helpful feedback on drafts of this article.

SFS is funded by the DARPA Composable High Assurance Trusted Systems program under contract #N66001-01-1-8927. Support also came from an NFS Graduate Fellowship and a USENIX Scholars Research Grant.