Self-certifying File System

by

David Mazières

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author ....
Department of Electrical Engineering and Computer Science
March 15, 2000

Certified by ....
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Self-certifying File System

by

David Mazières

Submitted to the Department of Electrical Engineering and Computer Science
on March 15, 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

No secure network file system has ever grown to span the Internet. Existing systems all lack adequate key management for security at a global scale. Given the diversity of the Internet, any particular mechanism a file system employs to manage keys will fail to support many types of use.

We propose separating key management from file system security, letting the world share a single global file system no matter how individuals manage keys. We present SFS, a secure file system that avoids internal key management. While other file systems need key management to map file names to encryption keys, SFS file names effectively contain public keys, making them *self-certifying pathnames*. Key management in SFS occurs outside of the file system, in whatever procedure users choose to generate file names.

Self-certifying pathnames free SFS clients from any notion of administrative realm, making inter-realm file sharing trivial. They let users authenticate servers through a number of different techniques. The file namespace doubles as a key certification namespace, so that people can realize many key management schemes using only standard file utilities. Finally, with self-certifying pathnames, people can bootstrap one key management mechanism using another. These properties make SFS more versatile than any file system with built-in key management.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

I would like to thank Frans Kaashoek for his immense help in building SFS and presenting its ideas, for undertaking the kind of grungy, low-level hacking few students can expect from professors, and for generally being such a wonderful advisor. I would also like to thank Michael Kaminsky, Chuck Blake, Kevin Fu, and Emmett Witchel for their contributions to the SFS software. Thanks also to Chuck for his emergency assistance with hardware, and to Kevin for getting a Verisign certificate and exploring the process. I thank Robert Morris for his help in analyzing various performance artifacts of NFS. I am grateful to David Black, the shepherd of our SOSP paper, for many suggestions on the design and the presentation of SFS. Butler Lampson, Ron Rivest, David Presotto, and the members of PDOS provided insightful comments on this work. I would also like to thank the building managers, Spaulding and Slye, for keeping my office so intolerably hot that I had to graduate quickly. Finally, I am deeply grateful to Mike Smith; without his help early on, I would never have gotten a chance to perform this work.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents SFS, a secure network file system designed to span the Internet. SFS provides one namespace for all files in the world. Users can access their files from any machine they trust, anywhere in the world. They can share files across organizational boundaries by merely exchanging file names. Like the web, anyone can set up an SFS server, any client can access any server, and any server can link or point to any other server. Thus, SFS is realistically deployable on a global scale, unlike any previous secure file system.

SFS provides strong security over untrusted networks, letting people share even sensitive files over the Internet. It resists eavesdropping and packet tampering attacks by encrypting and MACing (protecting from modification) client–server traffic. Careful engineering and choice of algorithms avoids any serious performance penalty from cryptography. SFS also prevents unauthorized users from accessing and modifying files; it performs cryptographically strong user authentication in a manner mostly transparent to the user. Most importantly, however, SFS also authenticates servers to clients.

Authenticating servers to clients poses the biggest challenge to global systems, particularly when users may access servers anonymously. In any global system, as in the web, one cannot hope to compile a complete list of all servers. Yet, any client must still be able to access any server. In fact, any user can potentially access any server at any time. Thus, a client must be capable of authenticating any server in the world, on the spot, without even necessarily having heard of it before.

SFS relies on public key cryptography to authenticate file servers. Every server has a public/private key pair. Clients use a server's public key to authenticate a secure channel to that server. Of course, this raises the problem of key management—namely how clients should get a server's public key in the first place. Unfortunately, no system has ever achieved key management at the scale of the Internet. Different situations call for different forms to key management. Any system with fixed key management can never hope to satisfy everybody's needs.

SFS therefore takes a new approach to key management. It provides global security without actually performing any key management. Instead, SFS embeds the equivalent of public keys in file names called *self-certifying pathnames*. This naming scheme pushes key management out of the file system into whatever process users choose for selecting file names. SFS consequently lets multiple key management schemes coexist. It furthermore goes to great lengths to make new schemes easy to implement. Thus, people can use the key management best suited to their needs, or even combine techniques to authenticate servers in ways not possible in traditional file systems.

SFS introduces several new mechanisms to make key management easy to implement. Self-certifying pathnames specify public keys, but as file names they can also be managed through ordinary file utilities. A mechanism called *agents* lets ordinary users plug external *certification programs* into the file system to authenticate servers. SFS also supports *secure symbolic links* between servers—like web links that can additionally specify public keys or how to get public keys. SFS supports highly-secure *read-only file systems* that can be served from untrusted machines and function as certification authorities. Finally, SFS allows secure data sharing between machines, and such shared data can be used to implement key management. Thus, SFS provides a great infrastructure with which to manage its own keys. Users can build powerful key management mechanisms using simple Unix shell scripts to access the file system.

SFS adds further flexibility by decoupling user authentication from the file system with a modular architecture. External programs authenticate users with protocols opaque to the file system software. User authentication programs communicate with the file system through well-defined RPC interfaces. Thus, programmers can easily replace them without touching the internals of the file system.

We implemented SFS focusing on three major goals: security, extensibility, and portability. We achieved portability by running in user space and speaking an existing network file system protocol (NFS [26]) to the local machine. As a result, the SFS client and server software run on most UNIX platforms. We sacrificed performance for portability in our implementation. Nonetheless, even from user-space, SFS performs comparably to NFS version 3 on application benchmarks. Several people access their home directories through SFS and perform all their work over it.

Chapter 2

Related work

SFS is the first file system to separate key management from file system security. No other file system embeds the equivalent of public keys in file names or lets users manage keys through the the file namespace. SFS is also the first file system to support both password authentication of servers and certification authorities. In this section, we relate SFS to other file systems and other secure network software.

2.1 File systems

AFS [13, 27, 28] is probably the most successful wide-area file system to date. We discuss AFS in detail, followed by a brief summary of other file systems.

AFS. AFS mounts all remote file systems under a single directory, `/afs` (an idea adopted by SFS). AFS does not provide a single global file system image, however; client machines have a fixed list of available servers (called *CellServDB*) that only a privileged administrator can update. AFS uses Kerberos [32] shared secrets to protect network traffic, and thus cannot guarantee the integrity of data from file systems on which users do not have accounts. Though AFS can be compiled to encrypt network communications to servers on which users have accounts, the commercial binary distributions in widespread use do not offer any secrecy. DFS [15] is a second generation file system, based on AFS, in which a centrally maintained database determines all available file systems.

To make the benefits of self-certifying pathnames more concrete, consider the following security conundrum posed by AFS. AFS uses password authentication to guarantee the integrity of remote files.[1] When a user logs into an AFS client machine, she uses her password and the Kerberos protocol to obtain a session key shared by the file server. She then gives this key to the AFS client software. When the user subsequently accesses AFS files, the client uses the shared key both to authenticate outgoing requests to the file server and to verify the authenticity of replies.

Because the AFS user knows her session key (a necessary consequence of obtaining it with her password), she knows everything she needs to forge arbitrary replies from the file server. In particular, if the user is malicious, she can pollute the client's disk cache, buffer cache, and name cache with rogue data for parts of the file system she should not have permission to modify.

When two or more users log into the same AFS client, this poses a security problem. Either the users must all trust each other, or they must trust the network, or the operating system must maintain separate file system caches for all users—an expensive requirement that, to the best of our knowledge, no one has actually implemented. In fairness to AFS, its creators designed the system for use on single-user workstations. Nonetheless, in practice people often set up multi-user AFS clients as dial-in servers, exposing themselves to this vulnerability.

Self-certifying pathnames prevent the same problem from occurring in SFS, because the pathnames themselves specify server public keys. Two users can both retrieve a self-certifying pathname using their passwords. If they end up with the same path, they can safely share the cache; they are asking for a server with the same public key. Since neither user knows the corresponding private key, neither can forge messages from the server. If, on the other hand, the users disagree over the file server's public key (for instance because one user wants to cause trouble), the two will also disagree on the names of files. They will end up accessing different files with different names, which the file system will consequently cache separately.

[1] Actually, AFS uses an insecure message authentication algorithm—an encrypted CRC checksum with a known polynomial. This problem is not fundamental, however.

Other file systems. The Echo distributed file system [4, 5, 17, 18] uses Taos's authentication infrastructure to achieve secure global file access without global trust of the authentication root. Clients need not go through the authentication root to access volumes with a common ancestor in the namespace hierarchy. However, the trust hierarchy has a central root implemented with DNS (and presumably requiring the cooperation of root name servers). Echo can short-circuit the trust hierarchy with a mechanism called "secure cross-links." It also has consistent and inconsistent versions of the file system protocol, while SFS for similar purposes uses both read-write and read-only file system protocols.

The Truffles service [23] is an extension of the Ficus file system [12] to operate securely across the Internet. Truffles provides fine-grained access control with the interesting property that a user can export files to any other user in the world, without the need to involve administrators. Unfortunately, the interface for such file sharing is somewhat clunky, involving the exchange of E-mail messages signed and encrypted with PEM. Truffles also relies on centralized, hierarchical certification authorities, naming users with X.500 distinguished names and requiring X.509 certificates for every user and every server.

WebFS [33] implements a network file system on top of the HTTP protocol. Specifically, WebFS uses the HTTP protocol to transfer data between user-level HTTP servers and an in-kernel client file system implementation. WebFS therefore allows the contents of existing URLs to be accessed through the file system. It also attempts to provide authentication and security through a protocol layered over HTTP; authentication requires a hierarchy of certification authorities.

2.2 Internet network security

SSL. SSL [11] is the most-widely deployed protocol for secure communication between web browsers and servers. Server authentication is based on SSL certificates—digitally signed statements that a particular public key belongs to a particular Internet domain name. To run a secure web server, a site must purchase a certificate from a widely trusted certification authority—for example, Verisign. When a browser connects to the server, the server sends back this certificate. The browser knows Verisign's public key and uses it to validate the certificate. If the certificate checks out, the browser knows it has the web server's real public key. It uses this key to set up a secure channel.

One can imagine a distributed file system consisting of a modified version of an existing file system such as NFS 3 running over SSL. We rejected this design because SSL's approach to key management is inappropriate for most file servers. Unclassified military networks, for instance, should not trust civilian certification authorities. Students setting up file servers should not need the cooperation of university officials with the authority to apply for certificates. Setting up a secure file server should be as simple and decentralized a process as setting up an ordinary, insecure web server.

We decided to purchase a certificate from Verisign to set up a secure web server. We were willing to pay Verisign's \$350 fee to conduct the experiment. To avoid involving university administrators, we decided not to apply for a certificate in the `mit.edu` domain. Instead, we purchased a domain of our own. This domain did not belong to a corporation, so Verisign required us to apply for a DBA ("Doing Business As") license at City Hall. To get a DBA we had to pay \$20 and show a driver's license, but City Hall neither verified our business's address nor performed any on-line checks to see if the name was already in use. Our business was not listed in the telephone directory, so Verisign could not call to perform an employment check on the person requesting the certificate. Instead this person had to fax them a notarized statement testifying that he was involved in the business. One week and \$440 later, we received a Verisign certificate for a single server.

While Verisign's certification procedure may seem cumbersome, the security of a certificate is only as good as the checks performed by the issuing authority. When a client trusts multiple certification authorities, SSL provides only as much security as the weakest one. Thus, SSL forces a trade-off between security and ease of setting up servers. SFS imposes no such trade-off. By pushing key management outside of the file system, SFS lets high- and low-grade certification schemes exist side-by-side. A user can access sensitive servers through Verisign without losing the ability separately to browse sites certified by a less trusted authority. More importantly, however, when users have passwords on servers, SRP gives them secure access without ever involving a certification authority.

Of course, as described in Section 3.4, SFS agents could actually exploit the existing SSL public key infrastructure to authenticate SFS servers.

IPsec. IPsec [16] is a standard for encrypting and authenticating Internet network traffic between hosts or gateways. IPsec specifies packet formats for encrypted data, but leaves the particulars of key management open-ended. Unfortunately, no global key management proposal has yet reached even the level of deployment of SSL certificates. Moreover, IPsec is geared towards security between machines or networks, and ill-suited to applications like SFS in which untrusted users participate in key management and sign messages cryptographically bound to session keys.

SPKI/SDSI. SPKI/SDSI [8, 24] is a key distribution system that is similar in spirit to SFS's egalitarian namespace and that could be implemented on top of SFS. In SPKI/SDSI, principals are public keys, and every principal acts as a certification authority for its own namespace. SFS effectively treats file systems as public keys; however, because file systems inherently represent a namespace, SFS has no need for special certification machinery—symbolic links do the job. SDSI specifies a few special roots, such as `Verisign!!`, which designate the same public key in every namespace. SFS can achieve a similar result by convention if clients all install symbolic links to certification authorities in their local root directories.

Chapter 3

Design

SFS's design has a number of key ideas. SFS names files with self-certifying pathnames that allow it to authenticate servers without performing key management. Through a modular implementation, SFS also pushes user authentication out of the file system. SFS itself functions as a convenient key management infrastructure, making it easy to implement and combine various key management mechanisms. Finally, SFS separates key revocation from key distribution, preventing flexibility in key management from hindering recovery from compromised keys. This section details the design of SFS.

3.1 Goals

SFS's goal of spanning the Internet faced two challenges: security and the diversity of the Internet. Attackers can easily tamper with network traffic, making strong security necessary before people can trust their files to a global file system. At the same time, SFS must satisfy a wide range of Internet users with different security needs. It is not sufficient for SFS to scale to many machines in theory—it must also satisfy the specific needs of diverse users on the Internet today. In short, SFS needs three properties to achieve its goals: a global file system image, security, and versatility.

3.1.1 Global file system image

SFS's goal of a single global file system requires that it look the same from every client machine in the world. It must not matter which client a person uses to access her files—a global file system should behave the same everywhere. Moreover, no incentive should exist for sites to subvert the global image by creating an "alternate" SFS (for instance, out of the need to have a different set of servers visible).

To meet this goal, we stripped the SFS client software of any notion of administrative realm. SFS clients have no site-specific configuration options. Servers grant access to users, not to clients. Users can have accounts on multiple, independently administered servers. SFS's global file system image then allows simultaneous access to all the servers from any client.

3.1.2 Security

SFS splits overall security into two pieces: *file system security* and *key management*. SFS proper provides only file system security. Informally, this property means that attackers cannot read or modify the file system without permission, and programs get the correct contents of whatever files they ask for. We define the term more precisely by enumerating the assumptions and guarantees that SFS makes.

SFS assumes that users trust the clients they use—for instance, clients must actually run the real SFS software to get its benefits. Users must also trust servers to store and return file data correctly. SFS servers are really principals that posses a particular private key. Thus, users much trust any machines with access to a file system's private key, and any administrators with access to those machines. To get practical cryptography, SFS additionally assumes computationally bounded adversaries and a few standard complexity-theoretic hardness conjectures. Finally, SFS assumes that malicious parties entirely control the network. Attackers can intercept packets, tamper with them, and inject new packets onto the network.

Under these assumptions, SFS ensures that attackers can do no worse than delay the file system's operation or conceal the existence of servers until reliable network communication is reestablished. SFS cryptographically enforces all file access control. Users cannot read, modify, delete, or otherwise tamper with files without possessing an appropriate secret key, unless anonymous access is explicitly permitted. SFS also cryptographically guarantees that results of file system operations come from the appropriate server or private key owner. Clients and read-write servers always communicate over a low-level secure channel that guarantees secrecy, data integrity, freshness (including replay prevention), and forward secrecy (secrecy of previously recorded encrypted transmissions in the face of a subsequent compromise). The encryption keys for these channels cannot be shortened to insecure lengths without breaking compatibility.

File system security in itself does not usually satisfy a user's overall security needs. Key management lets the user harness file system security to meet higher-level security goals. The right key management mechanism depends on the details of a user's higher-level goals. A user may want to access a file server authenticated by virtue of a pre-arranged secret password, or else the file system of a well-known company, or even the catalog of any reputable merchant selling a particular product. No key management mechanism satisfies all needs. Thus, SFS takes the approach of satisfying many key management mechanisms; it provides powerful primitives from which users can easily build a wide range of key management mechanisms.

3.1.3 Versatility

SFS should support as broad a range of uses as possible—from password-authenticated access to one's personal files to browsing well-known servers. In all cases, SFS must avoid unnecessary barriers to deployment. In particular, anyone with an Internet address or domain name should be able to create a new file server without consulting or registering with any authority.

SFS achieves versatility with three properties: an egalitarian namespace, a powerful set of primitives with which to implement key management, and modularity. Though SFS gives every file the same name on every client, no one controls the global namespace; everyone has the right to add a new server to this namespace.

$$\texttt{/sfs/}\overbrace{\texttt{sfs.lcs.mit.edu}}^{\textit{Location}}\texttt{:}\overbrace{\texttt{vefvsv5wd4hz9isc3rb2x648ish742h}}^{\textit{HostID}\text{ (specifies public key)}}\overbrace{\texttt{/pub/links/sfscvs}}^{\text{path on remote server}}$$

Figure 3-1: A self-certifying pathname

SFS's secure, global namespace also facilitates a broad array of key management schemes. One can implement many schemes by simply creating and serving files over SFS. SFS also lets users employ arbitrary algorithms during file name resolution to look up and certify public keys. Different users can employ different techniques to certify the same server; SFS lets them safely share the file cache.

Finally, SFS has a modular implementation. The client and server are each broken into a number of programs that communicate through well-defined interfaces. This architecture makes it easy to replace individual parts of the system and to add new ones—including new file system and user-authentication protocols. Several pieces of client functionality, including user authentication, occur in unprivileged processes under the control of individual users. Users therefore have a maximal amount of configuration control over the file system, which helps eliminate the need for clients to know about administrative realms.

3.2 Self-certifying pathnames

As a direct consequence of its design goals, SFS must cryptographically guarantee the contents of remote files without relying on external information. SFS cannot use local configuration files to help provide this guarantee, as such files would violate the global file system image. SFS cannot require a global authority to coordinate security either, as such an authority would severely limit versatility. Individual users might supply clients with security information, but this approach would make sharing a file cache very difficult between mutually distrustful users.

Without external information, SFS must obtain file data securely given only a file name. SFS therefore introduces *self-certifying pathnames*—file names that inherently specify all information necessary to communicate securely with remote file servers, namely a network address and a public key.

Every SFS file system is accessible under a pathname of the form `/sfs/`*Location* : *HostID*. *Location* tells an SFS client where to look for the file system's server, while *HostID* tells the client how to certify a secure channel to that server. *Location* can be either a DNS hostname or an IP address. To achieve secure communication, every SFS server has a public key. *HostID* is a cryptographic hash of that key and the server's *Location*. *HostID*s let clients ask servers for their public keys and verify the authenticity of the reply. Knowing the public key of a server lets a client communicate securely with it.

SFS calculates *HostID* with SHA-1 [9], a collision-resistant hash function:

$$\begin{aligned} \mathit{HostID} = \text{SHA-1}\,(&\text{“HostInfo”}, \mathit{Location}, \mathit{PublicKey}, \\ &\text{“HostInfo”}, \mathit{Location}, \mathit{PublicKey}) \end{aligned}$$

SHA-1 has a 20-byte output, much shorter than public keys. Nonetheless, finding any two inputs of SHA-1 that produce the same output is believed to be computationally intractable.[1] Thus, no computationally bounded attacker can produce two public keys with the same *HostID*; *HostID* effectively specifies a unique, verifiable public key. Given this scheme, the pathname of an SFS file system entirely suffices to communicate securely with its server.

Figure 3-1 shows the format of an actual self-certifying pathname. All remote files in SFS lie under the directory `/sfs`. Within that directory, SFS mounts remote file systems on self-certifying pathnames of the form *Location*:*HostID*. SFS encodes the 20-byte *HostID* in base 32, using 32 digits and lower-case letters. (To avoid confusion, the encoding omits the characters “l” [lower-case L], “1” [one], “0” and “o”.)

SFS clients need not know about file systems before users access them. When a user references a non-existent self-certifying pathname in `/sfs`, a client attempts to

[1]SFS actually duplicates the input to SHA-1. Any collision of the duplicate input SHA-1 is also a collision of SHA-1. Thus, duplicating SHA-1's input certainly does not harm security; it could conceivably help security in the event that simple SHA-1 falls to cryptanalysis.

contact the machine named by *Location*. If that machine exists, runs SFS, and can prove possession of a private key corresponding to *HostID*, then the client transparently creates the referenced pathname and mounts the remote file system there.

Self-certifying pathnames combine with automatic mounting to guarantee everyone the right to create file systems. Given an Internet address or domain name to use as a *Location*, anyone can generate a public key, determine the corresponding *HostID*, run the SFS server software, and immediately reference that server by its self-certifying pathname on any client in the world.

Key management policy in SFS results from the names of the files users decide to access. One user can retrieve a self-certifying pathname with his password. Another can get the same path from a certification authority. A third might obtain the path from an untrusted source, but want cautiously to peruse the file system anyway. SFS doesn't care why users believe this pathname, or even what level of confidence they place in the files. SFS just delivers cryptographic file system security to whatever file system the users actually name.

3.3 The `/sfs` directory

The SFS client breaks several important pieces of functionality out of the file system into unprivileged user agent processes. Every user on an SFS client runs an unprivileged agent program of his choice, which communicates with the file system using RPC. The agent handles authentication of the user to remote servers, prevents the user from accessing revoked *HostID*s, and controls the user's view of the `/sfs` directory. Users can replace their agents at will. To access a server running a new user authentication protocol, for instance, a user can simply run the new agent on an old client with no special privileges.

The SFS client maps every file system operation to a particular agent based on the local credentials of the process making the request.[2] The client maintains a different

[2]Typically each user has one agent, and requests from all of the user's processes get mapped to that agent. Users can run multiple agents, however. Additionally, an *ssu* utility allows a user to map operations performed in a particular super-user shell to her own agent.

`/sfs` directory for each agent, and tracks which self-certifying pathnames have been referenced in which `/sfs` directory. In directory listings of `/sfs`, the client hides pathnames that have never been accessed under a particular agent. Thus, a naïve user who searches for *HostID*s with command-line filename completion cannot be tricked by another user into accessing the wrong *HostID*.

SFS agents have the ability to create symbolic links in `/sfs` visible only to their own processes. These links can map human-readable names to self-certifying pathnames. When a user accesses a file not of the form *Location*:*HostID* in `/sfs`, the client software notifies the appropriate agent of the event. The agent can then create a symbolic link on-the-fly so as to redirect the user's access.

3.4 Server key management

Most users will never want to manipulate raw self-certifying pathnames. Thus, one must ask if SFS actually solves any problems for the average user, or if in practice it simply shifts the problems to a different part of the system. We address the question by describing numerous useful server key management techniques built on SFS. In every case, ordinary users need not concern themselves with raw *HostID*s.

Manual key distribution. Manual key distribution is easily accomplished in SFS using symbolic links. If the administrators of a site want to install some server's public key on the local hard disk of every client, they can simply create a symbolic link to the appropriate self-certifying pathname. For example, given the server `sfs.lcs.mit.edu`, client machines might all contain the link: `/lcs` → `/sfs/sfs.lcs.mit.edu:vefvsv5wd4hz9isc3rb2x648ish742hy`. Users in that environment would simply refer to files as `/lcs/`.... The password file might list a user's home directory as `/lcs/users/dm`.

Secure links. A symbolic link on one SFS file system can point to the self-certifying pathname of another, forming a secure link. In the previous example, the path `/lcs/pub/links/sfscvs` designates the file `/pub/links/sfscvs` on the server `sfs.lcs.mit.edu`. That file, in turn, might be a symbolic link pointing to the self-certifying pathname of server `sfscvs.lcs.mit.edu`. Users following secure links need not know anything about *HostID*s.

Secure bookmarks. When run in an SFS file system, the Unix *pwd* command returns the full self-certifying pathname of the current working directory. From this pathname, one can easily extract the *Location* and *HostID* of the server one is currently accessing. We have a 10-line shell script called *bookmark* that creates a link *Location* → `/sfs/`*Location*:*HostID* in a user's `~/sfs-bookmarks` directory. With shells that support the *cdpath* variable, users can add this `sfs-bookmarks` directory to their *cdpath*s. By simply typing "`cd` *Location*", they can subsequently return securely to any file system they have bookmarked.

Certification authorities. SFS certification authorities are nothing more than ordinary file systems serving symbolic links. For example, if Verisign acted as an SFS certification authority, client administrators would likely create symbolic links from their local disks to Verisign's file system: `/verisign` → `/sfs/sfs.verisign.com:r6ui9gwucpkz85uvb95cq9hdhpfbz4pe`. This file system would in turn contain symbolic links to other SFS file systems, so that, for instance, `/verisign/sfs.mit.edu` might point to `/sfs/sfs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69`.

Unlike traditional certification authorities, SFS certification authorities get queried interactively. This simplifies certificate revocation, but also places high integrity, availability, and performance needs on the servers. To meet these needs, we implemented a dialect of the SFS protocol that allows servers to prove the contents of public, read-only file systems using precomputed digital signatures. This dialect makes the amount of cryptographic computation required from read-only servers proportional to the file system's size and rate of change, rather than to the number of clients connecting. It also frees read-only servers from the need to keep any on-line copies of their private keys, which in turn allows read-only file systems to be replicated on untrusted machines.

Password authentication. SFS lets people retrieve self-certifying pathnames securely from remote servers using their passwords. Unfortunately, users often choose poor passwords. Thus, any password-based authentication of servers must prevent attackers from learning information they can use to mount an off-line password-guessing attack.[3]

Two programs, *sfskey* and *sfsauthd*, use the SRP protocol [36] to let people securely download self-certifying pathnames using passwords. SRP permits a client and server sharing a weak secret to negotiate a strong session key without exposing the weak secret to off-line guessing attacks. To use SRP, an SFS user first computes a one-way function of his password and stores it with the *sfsauthd* daemon running on his file server. *sfskey* then uses the password as input to SRP to establish a secure channel to the *sfsauthd*. It downloads the file server's self-certifying pathname over this channel, and has the user's agent create a link to the path in the `/sfs` directory.

In the particular user-authentication infrastructure we built (see Section 3.5), each user has his own public keys with which to authenticate himself. A user can additionally register an encrypted copy of his private key with *sfsauthd* and retrieve that copy along with the server's self-certifying pathname. The password that encrypts the private key is typically also the password used in SRP—a safe design because the server never sees any password-equivalent data.

Suppose a user from MIT travels to a research laboratory and wishes to access files back at MIT. The user runs the command "`sfskey add dm@sfs.lcs.mit.edu`". The command prompts him for a single password. He types it, and the command completes successfully. The user's agent then creates a symbolic link `/sfs/sfs.lcs.mit.edu` → `/sfs/sfs.lcs.mit.edu:vefvsv5wd4hz9isc3rb2x648ish742hy`. The user types "`cd /sfs/sfs.lcs.mit.edu`". Transparently, he is authenticated to `sfs.lcs.mit.edu` using a private key that *sfskey* just downloaded in encrypted form over an SRP-negotiated secure channel. The user now has secure access to his files back at MIT. The process involves no system administrators, no certification authorities, and no need for this user to have to think about anything like public

[3]Of course, an attacker can always mount an on-line attack by connecting to a server and attempting to "authenticate" a self-certifying pathname with a guessed password. We make such on-line attacks very slow, however. Moreover, an attacker who guesses 1,000 passwords will generate 1,000 log messages on the server. Thus, on-line password guessing attempts can be detected and stopped.

keys or self-certifying pathnames.

Forwarding pointers. SFS never relies on long-lived encryption keys for secrecy, only for authentication. In particular, an attacker who compromises a file server and obtains its private key can begin impersonating the server, but he cannot decrypt previously recorded network transmissions. Thus, one need not change a file server's public key preemptively for fear of future disclosure.

Nevertheless, servers may need to change their self-certifying pathnames (for instance if they change domain names). To ease the transition if the key for the old path still exists, SFS can serve two copies of the same file system under different self-certifying pathnames. Alternatively, one can replace the root directory of the old file system with a single symbolic link or forwarding pointer to the new self-certifying pathname.

Of course, if a self-certifying pathname change is precipitated by disclosure of the old private key, an attacker can serve rogue data to users instead of the correct forwarding pointer. As discussed in Section 3.5.3, a different mechanism is needed to revoke the pathnames of compromised private keys.

Certification paths. A user can give his agent a list of directories containing symbolic links, for example `~/sfs-bookmarks`, `/verisign`, `/verisign/yahoo`. When the user accesses a non-self-certifying pathname in `/sfs`, the agent maps the name by looking in each directory of the certification path in sequence. If it finds a symbolic link of the same name as the file accessed, it redirects the user to the destination of this symbolic link by creating a symbolic link on-the-fly in `/sfs`.

Existing public key infrastructures. On-the-fly symbolic link creation in `/sfs` can be used to exploit existing public key infrastructures. For example, one might want to use SSL [11] certificates to authenticate SFS servers, as SSL's certification model suits some purposes well. One can in fact configure an agent to generate self-certifying pathnames from SSL certificates. The agent might intercept every request for a file name of the form `/sfs/hostname.ssl`. It would contact `hostname`'s secure web server, download and check the server's certificate, and construct from the certificate a self-certifying pathname to which to redirect the user.

3.5 User authentication

While self-certifying pathnames solve the problem of authenticating file servers to users, SFS must also authenticate users to servers. As with server authentication, no single means of user authentication best suits all needs. SFS therefore separates user authentication from the file system. External software authenticates users through protocols of its own choosing.

On the client side, agents handle user authentication. When a user first accesses an SFS file system, the client delays the access and notifies his agent of the event. The agent can then authenticate the user to the remote server before the file access completes. On the server side, a separate program, the authentication server or "authserver," performs user authentication. The file server and authserver communicate with RPC.

The agent and authserver pass messages to each other through SFS using a (possibly multi-round) protocol opaque to the file system software. If the authserver rejects an authentication request, the agent can try again using different credentials or a different protocol. Thus, one can add new user authentication protocols to SFS without modifying the actual file system software. Moreover, a single agent can support several protocols by simply trying them each in succession to any given server.

If a user does not have an account on a file server, the agent will after some number of failed attempts decline to authenticate the user. At that point, the user will access the file system with anonymous permissions. Depending on the server's configuration, this may permit access to certain parts of the file system.

3.5.1 *sfsagent* and *sfsauthd*

This section describes the user authentication system we designed and built for SFS using the framework just described. Our system consists of an agent program called *sfsagent* and an authserver, *sfsauthd*.

One of the great advantages of self-certifying pathnames is the ease with which they let anyone establish a new file server. If users had to think about authenticating themselves separately to every new file server, however, the burden of user authentication would discourage the creation new servers. Thus, our goal was to make user authentication as transparent as possible to users of SFS.

All users have one or more public keys in our system. *sfsagent* runs with the corresponding private keys. When a client asks an agent to authenticate its user, the agent digitally signs an authentication request. The request passes through the client to server, which has *sfsauthd* validate it. *sfsauthd* maintains a database mapping public keys to user credentials. When it receives a valid request from the file server, *sfsauthd* replies with a set of Unix credentials—a user ID and list of group IDs.

sfsagent currently just keeps a user's private key in memory. However, we envisage a variety of more sophisticated agents. The agent need not have direct knowledge of any private keys. To protect private keys from compromise, for instance, one could split them between an agent and a trusted authserver using proactive security. An attacker would need to compromise both the agent and authserver to steal a split secret key. Alternatively, the agent might simply communicate through a serial port with a PDA that knows the key.

Proxy agents could forward authentication requests to other SFS agents. We hope to build a remote login utility similar to ssh [37] that acts as a proxy SFS agent. That way, users can automatically access their files when logging in to a remote machine. Authentication requests contain the self-certifying pathname of the server accessed by the user. They also contain a field reserved for the path of processes and machines through which a request arrives at the agent. Thus, an SFS agent can keep a full audit trail of every private key operation it performs.

3.5.2 User key management

sfsauthd translates authentication requests into credentials. It does so by consulting one or more databases mapping public keys to users. Because SFS is a secure file system, some databases can reside on remote file servers and be accessed through SFS itself. Thus, for example, a server can import a centrally-maintained list of users over SFS while also keeping a few guest accounts in a local database. *sfsauthd* automatically keeps local copies of remote databases; it can continue to function normally when it temporarily cannot reach the servers for those databases.

Each of *sfsauthd*'s public key databases is configured as either read-only or writable. *sfsauthd* handles a number of management tasks for users in writable databases. It allows them to connect over the network with *sfskey* and change their public keys, for example. It also lets them register SRP data and encrypted copies of their private keys for password authentication, as described in Section 3.4. To ease the adoption of SFS, *sfsauthd* can optionally let users who actually log in to a file server register initial public keys by typing their Unix passwords.

A server can mount a password guessing attack against a user if it knows her SRP data or encrypted private key. SFS makes such guessing attacks expensive, however, by transforming passwords with the eksblowfish algorithm [22]. Eksblowfish takes a cost parameter that one can increase as computers get faster. Thus, even as hardware improves, guessing attacks should continue to take almost a full second of CPU time per account and candidate password tried. Of course, the client-side *sfskey* program must invest correspondingly much computation each time it invokes SRP or decrypts a user's private key.

Very few servers actually need access to a user's encrypted private key or SRP data, however. *sfsauthd* maintains two versions of every writable database, a public one and a private one. The public database contains public keys and credentials, but no information with which an attacker could verify a guessed password. A server can safely export a public database to the world on an SFS file system. Other *sfsauthd*s can make read-only use of it. Thus, for instance, a central server can easily maintain the keys of all users in a department and export its public database to separately-administered file servers without trusting them.

3.5.3 Revocation

When a server's private key is compromised, its old self-certifying pathname may lead users to a fake server run by a malicious attacker. SFS therefore provides two mechanisms to prevent users from accessing bad self-certifying pathnames: key revocation and HostID blocking. Key revocation happens only by permission of a file server's owner. It automatically applies to as many users as possible. HostID blocking, on the other hand, originates from a source other than a file system's owner, and can conceivably happen against the owner's will. Individual users' agents must decide whether or not to honor blocked *HostID*s.

In keeping with its general philosophy, SFS separates key revocation from key distribution. Thus, a single revocation mechanism can revoke a *HostID* that has been distributed numerous different ways. SFS defines a message format called a key revocation certificate, constructed as follows:

$$\{\text{“PathRevoke”}, \mathit{Location}, K, \text{NULL}\}_{K^{-1}}$$

Revocation certificates are self-authenticating. They contain a public key, K, and must be signed by the corresponding private key, K^{-1}. “PathRevoke” is a constant. *Location* corresponds to the *Location* in the revoked self-certifying pathname. NULL simply distinguishes revocation certificates from similarly formated forwarding pointers. A revocation certificate always overrules a forwarding pointer for the same *HostID*.

When the SFS client software sees a revocation certificate, it blocks further access by any user to the *HostID* determined by the certificate's *Location* and K. Clients obtain revocation certificates in two ways: from servers and from agents. When SFS first connects to a server, it announces the *Location* and *HostID* of the file system it wishes to access. The server can respond with a revocation certificate. This is not a reliable means of distributing revocation certificates, but it may help get the word out fast about a revoked pathname. Alternatively, when a user first accesses a self-certifying pathname, the client asks his agent to check if the path has been revoked. At that point the agent can respond with a revocation certificate.

Revocation certificates might be used as follows. Verisign decides to maintain a directory called `/verisign/revocations`. In that directory reside files named by *HostID*, where each file contains a revocation certificate for the corresponding *HostID*.

Whenever a user accesses a new file system, his agent checks the revocation directory to look for a revocation certificate. If one exists, the agent returns it to the client software.

Because revocation certificates are self-authenticating, certification authorities need not check the identity of people submitting them. Thus, even someone without permission to obtain ordinary public key certificates from Verisign could still submit revocation certificates.

Of course, people who dislike Verisign are free to look elsewhere for revocation certificates. Given the self-authenticating nature of revocation certificates, however, an "all of the above" approach to retrieving them can work well—even users who distrust Verisign and would not submit a revocation certificate to them can still check Verisign for other people's revocations.

Sometimes an agent may decide a pathname has gone bad even without finding a signed revocation certificate. For example, even if a file system's owner has not revoked the file system's key, an agent may find that a certification authority in some external public key infrastructure has revoked a relevant certificate. To accommodate such situations, the agent can request HostID blocking from the client. This prevents the agent's owner from accessing the self-certifying pathname in question, but does not affect any other users.

Both revoked and blocked self-certifying pathnames become symbolic links to the non-existent file `:REVOKED:`. Thus, while accessing a revoked path results in a file not found error, users who investigate further can easily notice that the pathname has actually been revoked.

Chapter 4

Session protocols

This section describes the protocols by which SFS clients set up secure chennels to servers and users authenticate themselves to servers. We use quoted values to represent constants. K_C, K_S, and K_U designate public keys (belonging to a client, server, and user, respectively). K^{-1} designates the private key corresponding to public key K. Subscript K represents a message encrypted with key K, while subscript K^{-1} signifies a message signed by K^{-1}.

4.1 Key negotiation

When the SFS client software sees a particular self-certifying pathname for the first time, it must establish a secure channel to the appropriate server. The client starts by connecting (insecurely) to the machine named by the *Location* in the pathname. It requests the server's public key, K_S (Figure 4-1, step 2), and checks that the key in fact matches the pathname's *HostID*. If the key matches the pathname, the client knows it has obtained the correct public key.

Once the client knows the server's key, it negotiates shared session keys using a protocol similar to Taos [35]. To ensure forward secrecy, the client employs a short-lived public key, K_C (Figure 4-1, step 3), which it sends to the server over the insecure network. The client then picks two random key-halves, k_{C1} and k_{C2}; similarly, the server picks random key-halves k_{S1} and k_{S2}. The two encrypt and exchange their key-halves as shown in Figure 4-1.

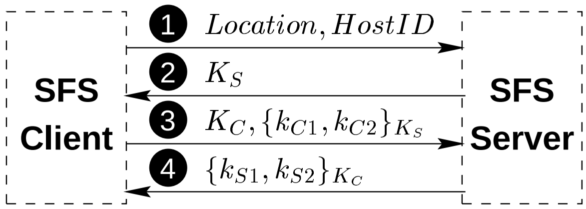


Figure 4-1: The SFS key negotiation protocol

The client and server simultaneously decrypt each other's key halves, overlapping computation to minimize latency. Finally, they compute two shared session keys—one for each direction—as follows:

$$\begin{aligned} k_{CS} &= \text{SHA-1}(\text{“KCS”}, K_S, k_{S1}, K_C, k_{C1}) \\ k_{SC} &= \text{SHA-1}(\text{“KSC”}, K_S, k_{S2}, K_C, k_{C2}) \end{aligned}$$

The client and server use these session keys to encrypt and guarantee the integrity of all subsequent communication in the session.

This key negotiation protocol assures the client that no one else can know k_{CS} and k_{SC} without also possessing K_S^{-1}. Thus, it gives the client a secure channel to the desired server. The server, in contrast, knows nothing about the client. SFS servers do not care which clients they talk to, only which users are on those clients. In particular, the client's temporary key, K_C, is anonymous and has no bearing on access control or user authentication. Clients discard and regenerate K_C at regular intervals (every hour by default).

4.2 User authentication

The current SFS agent and authserver rely on public keys for user authentication. Every user has a public key and gives his agent access to that key. Every authserver has a mapping from public keys to credentials. When a user accesses a new file system, the client software constructs an authentication request for the agent to sign. The client passes the signed request to the server, which asks the authserver to validate it.

SFS defines an *AuthInfo* structure to identify sessions uniquely:

$$\begin{aligned} \mathit{SessionID} &= \text{SHA-1}(\text{“SessionInfo”}, k_{SC}, k_{CS}) \\ \mathit{AuthInfo} &= \{\text{“AuthInfo”}, \text{“FS”}, \mathit{Location}, \mathit{HostID}, \mathit{SessionID}\} \end{aligned}$$

The client software also keeps a counter for each session to assign a unique sequence number to every authentication request.

When a user accesses a file system for the first time, the client initiates the user-authentication process by sending an *AuthInfo* structure and sequence number to the user's agent (see Figure 4-2). The agent returns an *AuthMsg* by hashing the *AuthInfo* structure to a 20-byte *AuthID*, concatenating the sequence number, signing the result, and appending the user's public key:

$$\begin{aligned} \mathit{AuthID} &= \text{SHA-1}(\mathit{AuthInfo}) \\ \mathit{SignedAuthReq} &= \{\text{“SignedAuthReq”}, \mathit{AuthID}, \mathit{SeqNo}\} \\ \mathit{AuthMsg} &= K_U,\ \{\mathit{SignedAuthReq}\}_{K_U^{-1}} \end{aligned}$$

The client treats this authentication message as opaque data. It adds another copy of the sequence number and sends the data to the file server, which in turn forwards it to the authserver. The authserver verifies the signature on the request and checks that the signed sequence number matches the one chosen by the client. If the request is valid, the authserver maps the agent's public key to a set of local credentials. It returns the credentials to the server along with the *AuthID* and sequence number of the signed message.

The server checks that the *AuthID* matches the session and that the sequence number has not appeared before in the same session.[1] If everything succeeds, the

[1] The server accepts out-of-order sequence numbers within a reasonable window to accommodate

server assigns an authentication number to the credentials, and returns the number to the client. The client tags all subsequent file system requests from the user with that authentication number. If, on the other hand, authentication fails and the agent opts not to try again, the client tags all file system requests from the user with authentication number zero, reserved by SFS for anonymous access.

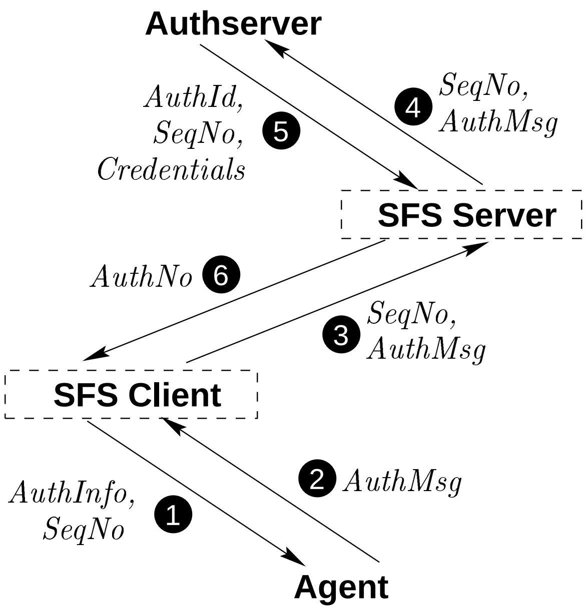


Figure 4-2: The SFS user authentication protocol

Sequence numbers are not required for the security of user authentication. As the entire user authentication protocol happens over a secure channel, all authentication messages received by the server must have been freshly generated by the client. Sequence numbers prevent one agent from using the signed authentication request of another agent on the same client. This frees the file system software from the need to keep signed authentication requests secret—a prudent design choice given how many layers of software the requests must travel through.

---

the possibility of multiple agents on the client returning out of order.

4.3 Cryptography

SFS makes three computational hardness assumptions. It assumes the ARC4 [14] stream cipher (allegedly the same as Rivest's unpublished RC4) is a pseudo-random generator. It assumes factoring is hard. Finally, it assumes that SHA-1 behaves like a random oracle [1].

SFS uses a pseudo random generator in its algorithms and protocols. We chose DSS's pseudo-random generator [10], both because it is based on SHA-1 and because it cannot be run backwards in the event that its state gets compromised. To seed the generator, SFS asynchronously reads data from various external programs (e.g., `ps`, `netstat`), from `/dev/random` (if available), from a `random_seed` file saved by the previous execution, and from a nanosecond (when possible) timer to capture the entropy of process scheduling. Programs that require users to enter a passphrase add both the keys typed and inter-keystroke timers as an additional source of randomness. All of the above sources are run through a SHA-1-based hash function [1] to produce a 512-bit seed. Because the external programs run in parallel and SFS reads from them asynchronously, SFS can efficiently seed the generator from all sources every time a program starts execution.

SFS uses the Rabin public key cryptosystem [34] for encryption and signing. The implementation is secure against adaptive chosen-ciphertext [2] and adaptive chosen-message [3] attacks. (Encryption is actually plaintext-aware, an even stronger property.) Rabin assumes only that factoring is hard, making SFS's implementation no less secure in the random oracle model than cryptosystems based on the better-known RSA problem. Like low-exponent RSA, encryption and signature verification are particularly fast in Rabin because they do not require modular exponentiation.

SFS uses a SHA-1-based message authentication code (MAC) to guarantee the integrity of all file system traffic between clients and read-write servers, and encrypts this traffic with ARC4. Both the encryption and MAC have slightly non-standard implementations. The ARC4 implementation uses 20-byte keys by spinning the ARC4 key schedule once for each 128 bits of key data. SFS keeps the ARC4 stream running for the duration of a session. It re-keys the SHA-1-based MAC for each message using 32 bytes of data pulled from the ARC4 stream (and not used for the purposes of encryption). The MAC is computed on the length and plaintext contents of each RPC message. The length, message, and MAC all get encrypted.

SFS's stream cipher is identical to ARC4 after the key schedule, and consequently has identical performance. SFS's MAC is slower than alternatives such as MD5 HMAC. Both are artifacts of the implementation and could be swapped out for more popular algorithms without affecting the main claims of the paper.

Chapter 5

Implementation

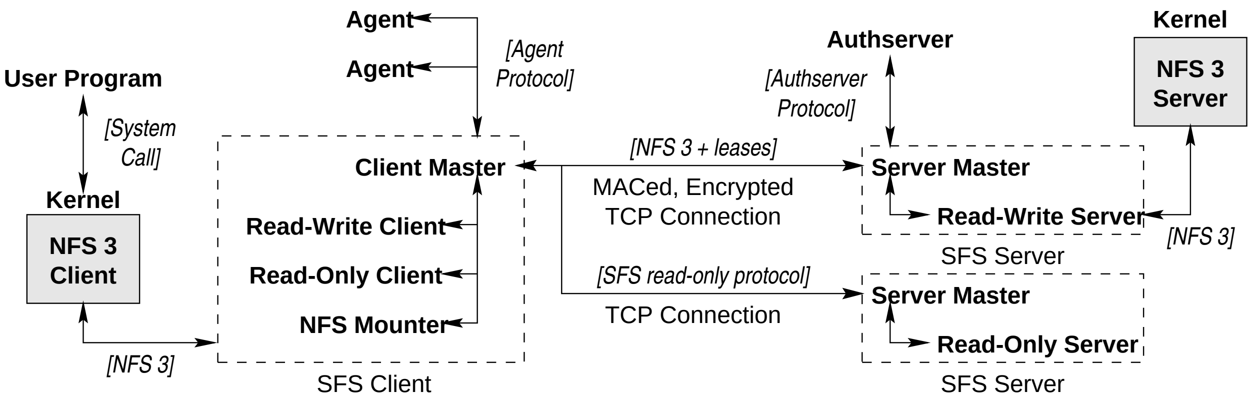


Figure 5-1: The SFS system components

Figure 5-1 shows the programs that comprise the SFS system. At the most basic level, SFS consists of clients and servers joined by TCP connections.

For portability, the SFS client software behaves like an NFS version 3 [6] server. This lets it communicate with the operating system through ordinary networking system calls. When users accesses files under SFS, the kernel sends NFS RPCs to the client software. The client manipulates the RPCs and forwards them over a secure channel to the appropriate SFS server. The server modifies requests slightly and tags them with appropriate credentials. Finally, the server acts as an NFS client, passing the request to an NFS server on the same machine. The response follows the same path in reverse.

5.1 Modularity and extensibility

Figure 5-1 reveals that a number of programs comprise the SFS system. All programs communicate with Sun RPC [30]. Thus, the exact bytes exchanged between programs are clearly and unambiguously described in the XDR protocol description language [31]. We also use XDR to define SFS's cryptographic protocols. Any data that SFS hashes, signs, or public-key encrypts is defined as an XDR data structure; SFS computes the hash or public key function on the raw, marshaled bytes. We use our own RPC compiler, specialized for C++, along with a new, asynchronous RPC library.

Breaking SFS into several programs helps the reliability, security, and extensibility of the implementation. Our RPC library can pretty-print RPC traffic for debugging, making it easy to understand any problems by tracing exactly how processes interact. We use SFS for our day-to-day computing, but have never run across a bug in the system that took more than a day to track down.

Within a machine, the various SFS processes communicate over UNIX-domain sockets. To authenticate processes to each other, SFS relies on two special properties of UNIX-domain sockets. First, one can control who connects to them by setting access permissions on directories. Second, one can pass file descriptors between processes over Unix-domain sockets. Several SFS daemons listen for connections on sockets in a protected directory, `/var/sfs/sockets`. A 100-line setgid program, *suidconnect*, connects to a socket in this directory, identifies the current user to the listening daemon, and passes the connected file descriptor back to the invoking process before exiting. The agent program connects to the client master through this mechanism, and thus needs no special privileges; users can replace it at will.

SFS's modularity facilitates the development of new file system protocols. On the client side, a client master process, *sfscd*, communicates with agents, handles revocation and forwarding pointers, and acts as an "automounter" for remote file systems. It never actually handles requests for files on remote servers, however. Instead, it connects to a server, verifies the public key, and passes the connected file descriptor to a subordinate daemon selected by the type and version of the server. On the server side, a server master, *sfssd*, accepts all incoming connections from clients. *sfssd* passes each new connection to a subordinate server based on the version of the client, the service it requests (currently fileserver or authserver), the self-certifying pathname it requests, and a currently unused "extensions" string.

A configuration file controls how client and server masters hand off connections. Thus, one can add new file system protocols to SFS without changing any of the existing software. Old and new versions of the same protocols can run alongside each other, even when the corresponding subsidiary daemons have no special support for backwards compatibility. As an example of SFS's protocol extensibility, we implemented a protocol for public, read-only file systems that proves the contents of file systems with digital signatures (See Chapter 8). As described in Section 3.4, read-only servers work well as SFS certification authorities. Implementing the read-only client and server required no changes to existing SFS code; only configuration files had to be changed.

5.2 NFS details

The SFS implementation was built with portability as a goal. Currently, the system runs on OpenBSD, FreeBSD, Solaris, Linux (with an NFS 3 kernel patch), and Digital Unix. Using NFS both to interface with the operating system on the client and to access files on the server makes portability to systems with NFS 3 support relatively painless.

The SFS read-write protocol, while virtually identical to NFS 3, adds enhanced attribute and access caching to reduce the number of NFS `GETATTR` and `ACCESS` RPCs sent over the wire. We changed the NFS protocol in two ways to extend the lifetime of cache entries. First, every file attribute structure returned by the server has a timeout field or lease. Second, the server can call back to the client to invalidate entries before the lease expires. The server does not wait for invalidations to be acknowledged; consistency does not need to be perfect, just better than NFS 3 on which SFS is implemented.

The NFS protocol uses numeric user and group IDs to specify the owner and group of a file. These numbers have no meaning outside of the local administrative realm. A small C library, *libsfs*, allows programs to query file servers (through the client) for mappings of numeric IDs to and from human-readable names. We adopt the convention that user and group names prefixed with "%" are relative to the remote file server. When both the ID and name of a user or group are the same on the client and server (e.g., SFS running on a LAN), *libsfs* detects this situation and omits the percent sign. We have modified the GNU file utilities to make use of *libsfs*, so that users can see and manipulate remote user and group names.

Using NFS has security implications. The SFS read-write server requires an NFS server. Running an NFS server can in itself create a security hole. NFS identifies files by server-chosen, opaque file handles (typically 32-bytes long). These file handles must remain secret; an attacker who learns the file handle of even a single directory can access any part of the file system as any user. SFS servers, in contrast, make their file handles publicly available to anonymous clients. SFS therefore generates its file handles by adding redundancy to NFS handles and encrypting them in CBC mode with a 20-byte Blowfish [29] key. Unfortunately, some operating systems use such poor random number generators that NFS file handles can potentially be guessed outright, whether or not one runs SFS.

One can avoid NFS's inherent vulnerabilities with packet filtering software. Several good, free packet filters exist and, between them, support most common operating systems. Sites with firewalls can also let SFS through the firewall without fearing such problems, so long as the firewall blocks NFS and portmap (which relays RPC calls) traffic. Many versions of Unix have a program called *fsirand* that randomizes NFS file handles. *fsirand* may do a better job of choosing file handles than a factory install of the operating system.

Another serious issue is that SFS effectively relays NFS 3 calls and replies to the kernel. During the course of developing SFS, we found and fixed a number of client and server NFS bugs in Linux, OpenBSD, and FreeBSD. In many cases, perfectly valid NFS messages caused the kernel to overrun buffers or use uninitialized memory. An attacker could exploit such weaknesses through SFS to crash or break into a machine running SFS. We think the low quality of most NFS implementations constitutes the biggest security threat to SFS.

The SFS client creates a separate mount point for each remote file system. This lets different subordinate daemons serve different file systems, with each subordinate daemon exchanging NFS traffic directly with the kernel. Using multiple mount points also prevents one slow server from affecting the performance of other servers. It ensures that the device and inode number fields in a file's attribute structure uniquely identify the file, as many file utilities expect. Finally, by assigning each file system its own device number, this scheme prevents a malicious server from tricking the *pwd* command into printing an incorrect path.

All NFS mounting in the client is performed by a separate NFS mounter program called *nfsmounter*. The NFS mounter is the only part of the client software to run as root. It considers the rest of the system untrusted software. If the other client processes ever crash, the NFS mounter takes over their sockets, acts like an NFS server, and serves enough of the defunct file systems to unmount them all. The NFS mounter makes it difficult to lock up an SFS client—even when developing buggy daemons for new dialects of the protocol.

5.3 Automounting in place

The SFS client creates self-certifying pathnames on the fly. Moreover, as discussed above, each remote file system has a separate mount point. Thus, SFS must automatically mount remote file systems as they are referenced, a process known as "automounting."

Several artifacts of Unix NFS implementations complicate the implementation of an automounter. When a user first references a self-certifying pathname, the kernel generates an NFS `LOOKUP` RPC. The automounter cannot immediately reply to this RPC, because it must first create the mount point for the new file system. However, it cannot create a mount point on the same file name, either, because client NFS implementations typically hold exclusive locks on the parent directory while a `LOOKUP` RPC is pending.

Worse yet, the SFS automounter cannot immediately create a mount point when it sees a self-certifying pathname. It must first perform a DNS lookup on the *Location* in the pathname, establish a TCP connection to the server, query the server for its dialect, and finally pass the connection off to the appropriate subsidiary daemon. NFS client implementations, in order to avoid flooding servers with retransmissions, typically lock NFS mount points and stop issuing new requests when the server takes too long to reply to an old request. Thus, the SFS automounter cannot sit on a `LOOKUP` request for a self-certifying pathname too long, or it will end up blocking all file accesses to the `/sfs` directory.

Previous NFS automounters such as automount [7] and *amd* [21] have taken the approach of mounting file systems outside of the automounter's directory and redirecting users with symbolic links. Thus, for instance, *amd* might serve a directory `/home`. When it sees a `LOOKUP` for a file named `/home/am2`, it would mount the corresponding file system somewhere else (for instance on `/a/amsterdam/u2`), then return from the `LOOKUP` RPC with a symbolic link, `/home/am2` → `/a/amsterdam/u2`.

Unfortunately, the traditional approach to automounting has two serious drawbacks. First, the Unix *pwd* command no longer returns the correct answer—in the *amd* example, it would return `/a/amsterdam/u2` rather than the pathname needed to automount the file system in the future, `/home/am2`. Second, when a file system takes too long to mount (or fail to mount—when the server is unavailable), the automounter ends up delaying the `LOOKUP` RPC too long, the client locks the mount point, and access to other, working, mounted file systems gets delayed. Solaris and Linux have each individually addressed this problem by building part of the automounter into the operating system kernel, but their solutions are incompatible with each other and other operating systems, and thus could not be used by SFS which strives to be portable.

SFS solves these automounter problems with two tricks. First, it tags *nfsmounter*, the process that actually makes the mount system calls, with a reserved group ID. This lets *sfscd* differentiate between NFS RPCs generated on behalf of a mount system call, and those issued by other root processes. Second, *sfscd* creates a number of special "`.mnt`" mount points on directories with names of the form `/sfs/.mnt/0/`, `/sfs/.mnt/1/`, ... *sfscd* never delays a response to a `LOOKUP` RPC on a self-certifying pathname. Instead, it returns a symbolic link redirecting the user to another symbolic link in one of the `.mnt` file systems, and there it delays the result of a `READLINK` RPC.

Meanwhile, as the user's file system request is being redirected to a `.mnt` file system, *sfscd* actually mounts the remote file system on the self-certifying pathname. Because *nfsmounter*'s NFS RPCs are tagged with a reserved group ID, *sfscd* can respond differently to them—giving *nfsmounter* a different view of the file system from the user's. Thus, while users referencing the self-certifying pathname see a symbolic link to `/sfs/.mnt/`..., *nfsmounter* sees an ordinary directory on which it can mount the remote file system. Once the mount succeeds, *sfscd* lets the user see the directory, and responds to the pending `READLINK` RPC redirecting the user to the self-certifying pathname that is now a directory.

5.4 Asynchronous I/O and RPC Libraries

All the daemons comprising the SFS system have an event-driven architecture. They are built around a new, non-blocking I/O library called *libasync*. SFS's daemons avoid performing operations that may block. When a function cannot complete immediately, it registers a callback with *libasync*, to be invoked when a particular asynchronous event occurs. *libasync* supports callbacks when file desctiprots become ready for reading or writing, when child processes exit, when a process receives signals, and when the clock passes a particular time. A central dispatch loop polls for such events to occur through the system call *select*—the only system call SFS makes that ever blocks.

Two complications arise from event-driven programming in a language like C or C++. First, in languages that do not support closures, it can be inconvenient to

```
class foo : public bar {
  /* ... */
};

void
function ()
{
  ref<foo> f = new refcounted<foo> (/* ... */);
  ptr<bar> b = f;
  f = new refcounted<foo> (/* ... */);
  b = NULL;
}
```

Figure 5-2: Example usage of reference couted pointers

bundle up the necessary state one must preserve to finish an operation in a callback. Second, when an asynchronous library function takes a callback and buffer as input and allocates memory for its results, the function's type signature does not make clear what code is responsible for freeing which memory when. Both complications easily lead to programming errors.

libasync makes asynchronous library interfaces less error-prone through aggressive use of C++ templates. A function *wrap* produces callback objects through a technique much like function currying: *wrap* bundles up a function pointer and some initial arguments to pass the function, and it returns a function object taking the function's remaining arguments. In other words, given a function:

```
res_t function (arg1_t, arg2_t, arg3_t);
```

a call to `wrap (function, a1, a2)` produces a function object with type signature:

```
res_t callback (arg3_t);
```

This *wrap* mechanism permits convenient bundling of code and data into callback objects in a type-safe way.

libasync also supports reference-counted garbage collection, avoiding the programming burden of tracking whether the caller or callee is responsible for freeing dynamically allocated memory for every given library function. Two template types offer reference counted pointers to objects of type T—ptr<T> and ref<T>. ptr and ref behave identically and can be assigned to each other except that a ref cannot be NULL. One can allocate a reference counted version of any type with the template type refcounted<T>, which takes the same constructor arguments as type T. Figure 5-2 shows an example use of reference-counted garbage collection.

SFS also uses a new asynchronous RPC library, *arpc*, built on top of *libasync*, and a new RPC compiler *rpcc*. *rpcc* compiles Sun XDR data structures into C++ data structures. Rather than directly output code for serializing the structures, however, *rpcc* uses templates and function overloading to produce a generic way of traversing data structures at compile time. Serialization of data in RPC calls is one application of this traversal mechanism, but SFS uses the mechanism in several other places. For instance, to protect NFS file handles, the server must encrypt them. Rather than hand-code one file handle encryption/decryption function for each of the 21 NFS 3 argument and return types, SFS does it in a single place, using the RPC traversal function to locate all file handles automatically.

Chapter 6

Agent implementation

Figure 6-1 shows the implementation of the SFS user agent. As described in Section 3.3, SFS agents serve three functions: authenticating users to remote servers, dynamically translating human-readable file names to self-certifying pathnames, and preventing the use of self-certifying pathnames corresponding to compromised private keys.

6.1 Agent components

The SFS agent's functionality is broken into two programs, *sfsagent* and *sfskey*. *sfsagent* is a long-running daemon that persists for the duration of a user's session. It performs the agent's three functions as needed. For user authentication, the current implementation simply holds user private keys in memory. Users might instead wish to store authentication keys in dedicated hardware such as a smart card, or have a single master process control authentication across multiple clients. To facilitate multiple implementations, therefore, *sfsagent* is as simple as possible. It totals under 1,000 of code.

The bulk of a user's key management needs are actually implemented in the command-line utility *sfskey*. *sfskey* controls and configures *sfsagent*. It fetches and decrypts private keys for user-authentication and sets up dynamic server authentication and revocation checking. *sfskey* can communicate directly with the SFS client

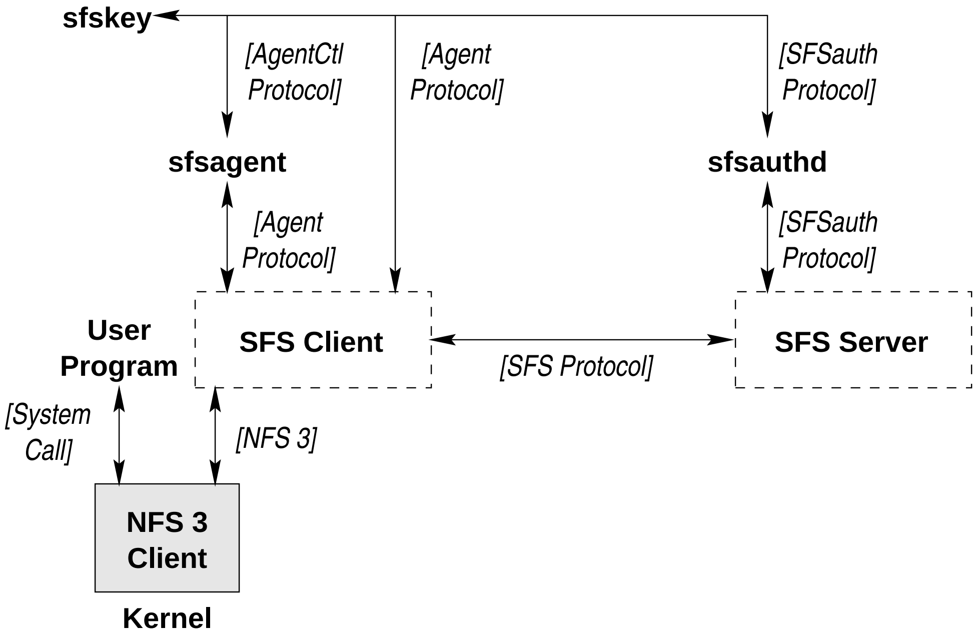
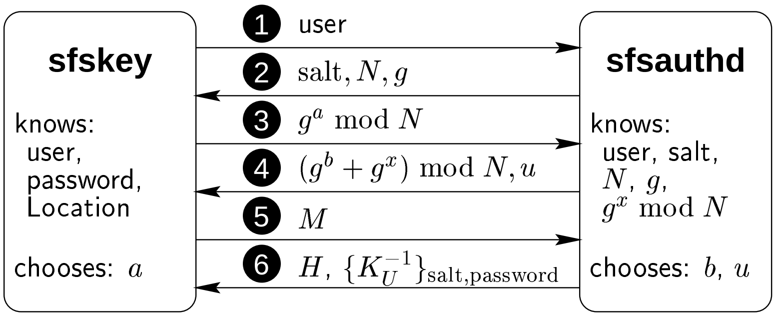


Figure 6-1: SFS agent implementation

software (for instance to kill the user's agent on logout). It also communicates directly with servers to let users fetch private keys or change their public keys.

A user typically runs *sfsagent* at login time. *sfsagent* connects to the client master daemon, *sfscd*, using *suidconnect* to achieve authenticated IPC (as described in Section 5.1). *sfscd* subsequently calls into *sfsagent* in response to file system operations by the user, for instance when the user must be authenticated to a new remote file server. Once the agent has started, the user configures it with *sfskey*. *sfskey* obtains an authenticated connection to *sfsagent* through *sfscd*, which ensures that one user cannot talk to another user's *sfsagent*.

sfsagent carries out user authentication with the protocol in Section 4.2. It maintains a list of private keys. Every time the user accesses a new server, *sfsagent* tries each private key in succession until the server accepts one. If the server refuses them all, *sfsagent* gives up and tells the SFS client to let the user access the file system with anonymous permissions. Each private key can have an expiration time associated with it, after which *sfsagent* automatically wipes the key's value from memory.



$$
\begin{aligned}
x &= \text{EKSBhash}_{|N|}(\text{salt}, \text{password}, \mathit{Location}) \\
S &= g^{ab+xub} \\
M &= \text{SHA-1}(\mathit{AuthID}, N, g, \text{user}, \text{salt}, g^a \bmod N, g^b \bmod N, S) \\
H &= \text{SHA-1}(\mathit{AuthID}, g^a \bmod N, g^b \bmod N, S)
\end{aligned}
$$

Figure 6-2: SRP protocol for mutual authentication of a secure channel with a user-chosen password. *sfskey* knows the user's password and chooses a random parameter $a \in \mathbf{Z}_N^*$. *sfsauthd* knows a one-way function of the user's password, $g^x \bmod N$ (which it keeps secret), and chooses two random parameters $b, u \in \mathbf{Z}_N^*$. During the protocol, *sfskey* verifies that, with probability greater than $1 - 2^{-64}$, N is prime and g is a generator of $\mathbf{Z}_N^*$.

6.2 Password authentication

sfskey employs user-chosen passwords for two purposes: to authenticate users to servers and to authenticate servers to users. It performs both functions with the same password, so that users need only type a password once to access a given server. Internally, user and server authentication are handled quite differently, however. User authentication takes place through the public key protocol of Section 4.2; passwords serve to obtain a user's private key before invoking this protocol. Server authentication consists of creating a symbolic link that maps a human-readable name to a self-certifying pathname; here passwords let *sfskey* securely download a server's *HostID* without fear of an attacker impersonating the server.

Because users often choose poor passwords, any password authentication mechanism should prevent an attacker from repeatedly guessing passwords from a dictionary of likely choices off-line. For example, though *sfskey* can retrieve an encrypted private key from a file and decrypt it with the user's password, such a file should not lie within SFS. The user needs the private key to access any protected files—thus the key itself, if stored on SFS, would have to reside in a world-readable file. An attacker could therefore downloaded the encrypted key and guess passwords off-line, on his own hardware, limited only by the computational cost of each guess.

To prevent off-line attacks, *sfskey* uses the SRP protocol [36] to establish password-authenticated secure channels. Users run *sfskey* to register their passwords with the *sfsauthd* daemon on one or more file servers. To register a user, *sfskey* chooses for the user a random, 128-bit salt, a prime number N such that $(N-1)/2$ is also prime, and a generator g of $\mathbf{Z}_N^*$. It then computes a number $x \in \mathbf{Z}_N^*$ by hashing the user's password, the salt, and the server's name with the cost-parameterizable eksblowfish algorithm [22]. *sfskey* then gives *sfsauthd* the salt, N, g, $g^x \bmod N$, and an encrypted copy of the user's private key. *sfsauthd* stores the information in a persistent database analogous to the Unix password file.

To invoke password authentication, the user types "`sfskey add` ***`user`***`@`***`Location`***". *sfskey* then establishes a secure channel to the server named by *Location* with the protocol of section 4.1. However, *sfskey* does not check the server's public key against any *HostID*, because the user has only supplied a *Location*. Instead, to authenticate themselves to each other, *sfskey* and *sfsauthd* engage in the protocol depicted in Figure 6-2. This protocol allows mutual authentication with no known way for an attacker impersonating one of the parties to learn any information useful for mounting an off-line password guessing attack. When the protocol is over, *sfsauthd* gives *sfskey* an encrypted copy of the user's private key. *sfskey* decrypts the private key with the same password and salt, and hands the key to *sfsagent*. Finally, *sfskey* calculates *HostID* from the key negotiation it used to set up a secure channel in the first place, and it creates a symbolic link:

$$\texttt{/sfs/}\mathit{Location} \rightarrow \mathit{Location}:\mathit{HostID}$$

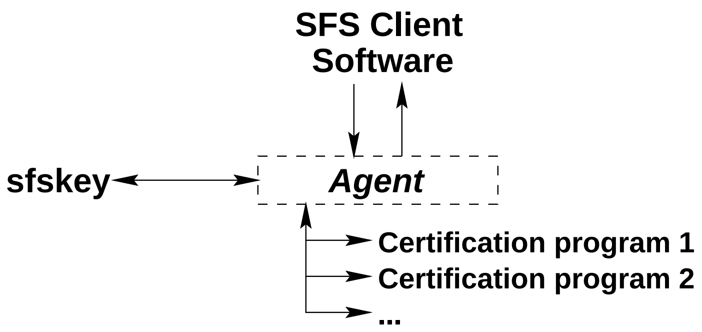


Figure 6-3: Certification programs

6.3 Dynamic server authentication

As mentioned in Section 3.3, each user has a different view of the `/sfs` directory. *sfsagent* can create symbolic links in `/sfs` on-the-fly. When a user accesses a file other than a self-certifying pathname in `/sfs`, *sfscd* notifies the user's agent of the event, giving the agent a chance to create a symbolic link and redirect the user's access. This mechanism can dynamically map human-readable names to self-certifying pathnames.

sfsagent has no built-in interpretation of non-self-certifying names in `/sfs`. Instead, it maps those names to other pathnames by invoking arbitrary external *certification programs*, as shown in Figure 6.3. In contrast to most systems, which manage keys in a monolithic piece of software, certification programs represent a toolkit approach to key management. Users plug small utilities together to manage keys in a style reminiscent of Unix command pipelines.

Using ceritificaton programs, powerful key management mechanisms often have implementations short enough to be typed at the command prompt. Different mechanisms can be cascaded or bootstrapped over one another to manage keys in ways not previously possible. A number of commands and utilities developed for other purposes can also be used to manage keys. Most importantly, however, certification programs can exploit the security of the file system to access remote data securely. Thus, few certification programs need even contain any cryptography. Chapter 7 will demonstrate these benefits with a number of examples.

Each certification program is a 4-tuple of the form:

$$\langle \textit{suffix},\ \textit{filter},\ \textit{exclude},\ \textit{program}\ [\textit{arg}\ldots] \rangle$$

suffix, *filter*, and *exclude* are all optional and can be left empty. *suffix* partitions the namespace of the `/sfs` directory into links managed by separate programs. If a certification program's *suffix* is non-empty, the program will only be run to look up names ending in *suffix*, and *sfsagent* will strip *suffix* from names before any further processing. *filter* and *exclude* are regular expressions to control the scope of a certification program. When *filter* is specified, the agent will only invoke the certification program on names containing the regular expression. Analogously, when *exclude* is specified, the agent will not invoke the certification program on any name containing *exclude*.

Finally, *program* is the external program to run, and the initial arguments to give the program. The final argument to the program will be the actual name to look up. If a certification program exits successfully and prints something to its standard output, the agent will use the program's output as the contents of the dynamically created symbolic link. If not, the agent will continue down the list of certification programs.

As a simple example, suppose a directory `/mit` contains symbolic links to the self-certifying pathnames of various SFS servers around MIT. A user might want to map names of the form `/sfs/`*host*`.mit` to `/mit/`*host*, but exclude any *host* ending `.lcs`. The following certification program would accomplish this task:

⟨`.mit`, , `\.lcs$`, `sh -c 'test -L /mit/$0 && echo /mit/$0'`⟩.

The *suffix* `.mit` applies this certification program to names in `/sfs` ending `.mit`. An empty *filter* signifies the certification program applies to all names. The *exclude* regular expression excludes any names ending `.lcs`. Finally, *program* consists of three arguments, `sh`, `-c`, and a short shell script. The agent will supply the name being looked up as the final argument, which the shell calls `$0`. The script simply checks for the existence of a symbolic link in `/mit`. If one exists, it prints the path and exits successfully; otherwise, it exits with non-zero status, indicating failure.

6.4 Server key revocation

As with dynamic server authentication, *sfsagent* uses external programs to handle server key revocation. *sfskey* gives *sfsagent* a list of *revocation programs* to run on each self-certifying pathname the user accesses. Each entry in the list is a 4-tuple of the form:

$$\langle block,\ filter,\ exclude,\ program\ [arg \ldots]\rangle$$

block is a boolean value that enables HostID blocking in addition to key revocation (see Section 3.5.3). *filter* and *exclude* are regular expression filters, as with certification programs, that here get applied to the *Location* part of self-certifying pathnames. *program* is the program to run to check the validity of a self-certifying pathname. The pathname's *HostID* is appended to the program's arguments.

A revocation program can write a self-authenticating revocation certificate to standard output. If it does so, the agent uploads the certificate to *sfscd*, which subsequently blocks all access to files under that pathname by any user. If the program exits successfully but does not output a revocation certificate, then, in the case that the *block* flag is set, the agent triggers HostID blocking, preventing its own user from accessing the self-certifying pathname without affecting other users.

Because revocation programs may need to access files in SFS, there is a potential for deadlock. For instance, a revocation program cannot synchronously check the validity of Verisign's self-certifying pathname while also searching for revocation certificates under `/verisign` (the symbolic link to Verisign's hypothetical server in Section 3.4). Thus, *sfskey* also gives *sfsagent* a "*norevoke list*" of *HostID*s not to check synchronously. *HostID*s in the norevoke list can still be revoked by uploading a revocation certificate to *sfscd*, but this is not guaranteed to happen before the user accesses the revoked pathname.

6.5 Agent identities

Agents are completely unprivileged and under the control of individual users. Users can replace their agents at will. The current agent implementation stores multiple private keys for authentication to more than one SFS server. However, users needing greater flexibility can also run several agent programs simultaneously.

The SFS client software maps each file system request to a particular agent. It does so not simply through a process's user ID, but through a more generalized notion called the agent ID or *aid*. Ordinarily, a process's *aid* is the same as its 32-bit user ID. Depending on a process's group list, however, the process can have a different 64-bit *aid* with the same low-order 32-bits as its user ID. System administrators set a range of reserved group IDs in the main SFS configuration file, and these groups mark processes as belonging to different *aid*s. A 150-line setuid root program *newaid* lets users switch between reserved group IDs to spawn processes under new *aid*s.

*aid*s let different processes run by the same local user access remote files with different credentials—for instance as a user and superuser. Conversely, a user may want to become root on the local machine yet still access remote files as himself. Root processes therefore can be assigned any *aid* of any user (i.e. with any low-order 32-bits). A utility *ssu* lets users become local superuser without changing *aid*s. *ssu* is a wrapper around the setuid Unix utility *su*, and thus does not itself need any special privileges.

Chapter 7

Key management cookbook

By invoking external certification programs to map server names to self-certifying pathnames, *sfsagent* lets users employ virtually arbitrary key management algorithms. More importantly, however, *sfsagent* actually facilitates the construction of new mechanisms, letting users easily meet unanticipated key management needs. This chapter describes a number of SFS certification programs. The simplicity of the implementations and their use of the file system convey how effective a key management infrastructure SFS is for itself.

7.1 Existing key management infrastructures

SFS can exploit any existing key management infrastructure that generates secure channels from the names of servers. To date, the most widely deployed such infrastructure is SSL [11]—the protocol underlying secure HTTP. The text-mode web browser *lynx*, if compiled with SSL support, will download a document from a secure web server using SSL to guarantee integrity.

SSL-enabled web servers can manage SFS server keys. For example, one might distribute self-certifying pathnames from secure URLs of the form `https://`*Host*`/sfspath.txt`. The command `lynx --source https://`*Host*`/sfspath.txt` will retrieve such pathnames and print them to standard output. Thus, the following command can be used to map pathnames of the form `/sfs/`*Host*`.ssl` to self-certifying

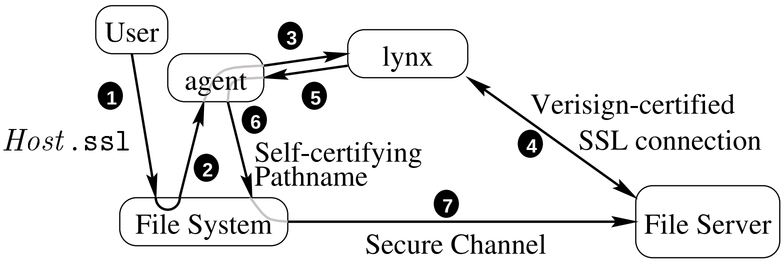


Figure 7-1: Flow of events when *sfsagent* fetches HostIDs with SSL.

pathnames retrieved over SSL:

```
sfskey certprog -s ssl \
     sh -c 'lynx --source https://$0/sfspath.txt'
```

The command *sfskey certprog* registers a new dynamic server authentication program with *sfsagent*. The argument `-s ssl` specifies that the authentication program applies to file names in `/sfs` with a *suffix* of `.ssl`. Finally, the unix command

```
sh -c 'lynx --source https://$0/sfspath.txt' Host
```

runs the *lynx* command between quotes, substituting *Host* for `$0`.

Figure 7-1 illustrates the flow of events when the user references a pathname of the form `/sfs/`*Host*`.ssl`. The file system on the client side blocks the request, and calls into the user's agent. The agent invokes *lynx* to download the server's HostID over a secure SSL connection. Finally, the agent returns a symbolic link *Host*`.ssl` → *Host*:*HostID*. The file system then unblocks the user's reference and lets the symbolic link redirect it to the server's self-certifying pathname.

SFS can similarly exploit other key management infrastructures. For example, the following commend allows users to manage SFS server keys using Kerberos:

```
sfskey certprog -s krb sh -c 'rsh -x $0 sfskey hostid -'
```

The command `rsh -x` *Host program [arg...]* establishes a secure connection to server *Host* with Kerberos V, runs *program* on the server, and sends the results back over the secure channel. The command `sfskey hostid -` prints the local machine's self-certifying pathname. Thus, the *sfskey certprog* command above intercepts references to pathnames of the form `/sfs/`*Host*`.krb` and maps them to the output of using Kerberos to run `sfskey hostid -` on *Host*.

The same can be done to exploit an ssh key management infrastructure in SFS. Here is the ssh equivalent of the Kerberos command above, mapping names of the form `/sfs/`*Host*`.ssh` to self-certifying pathnames retrieved over ssh:

```
sfskey certprog -s ssh \
     sh -c 'ssh -x -a -o"FallBackToRsh no" -o"BatchMode yes" \
         -o"StrictHostKeyChecking yes" $0 sfskey hostid -'
```

7.2 Certification paths

Section 3.4 introduced the notion of certification paths—prioritized lists of certification authorities to search for self-certifying pathnames. Because SFS certification authorities are nothing more than ordinary file systems, certification paths can be implemented quite simply.

The SFS distribution includes a small C program called *dirsearch* that searches a list of directories for a file of a given name. The usage is:

`dirsearch` *dir1 [dir2...] name*

dirsearch goes through the directories *dir1*, *dir2*, ... in order and prints the full pathname of the first file named *name* that it runs across.

dirsearch lets users easily implement certification directories. For example to search the directories `~/sfs-bookmarks`, `/verisign`, and `/verisign/yahoo`, the user could enter the following command:

```
sfskey certprog dirsearch ~/sfs-bookmarks \
     /verisign /verisign/yahoo
```

Then, if `/verisign` contains a symbolic link named `MIT` and `~/sfs-bookmarks` does not, any reference to the pathname `/sfs/MIT` would transparently produce a symbolic link `/sfs/MIT` → `/verisign/MIT`.

```
#!/bin/sh

# Make known_hosts directory if it does not exist
linkdir=${HOME}/.sfs/known_hosts
test -d $linkdir || mkdir -p $linkdir || exit 1

# Use link in known_hosts directory if one exists
dirsearch -l $linkdir $1 && exit 0

# Retrieve and store pathname (insecurely, for the first time)
srvpath="`sfskey hostid $1`"
test "$srvpath" || exit 1
ln -s "/sfs/$srvpath" "$linkdir/$1"

# Print self-certifying pathname just retrieved
exec dirsearch -l $linkdir $1
```

Figure 7-2: *mmi.sh*: Script to perform ssh-like key management in SFS.

7.3 ssh-like key management

Ssh has made remote login security available to an unprecedented number of people. One of the main reasons for ssh's success was that it sacrificed a little bit of security to free users from the burden of key management. Ssh leaves users open to a man-in-the-middle attack the first time they access any given server. It records server public keys for subsequent accesses, however, so that an attacker must intercept a user's very first access to a server to impersonate that server. Despite its weakness, ssh offers better security than previous unencrypted remote login utilities, while still allowing users to connect to any server from any client.

In any situation where users have accounts on servers, the password protocol of Section 6.2 should let SFS users access the servers without man-in-the-middle vulnerabilities. Nonetheless, ssh's very popular key management model may still prove useful for anonymous access to servers with low security needs.

Figure 7.3 shows *mmi.sh*, a Unix shell script that implements ssh-like key management for ssh. One can use the script with a command like:

```
sfskey certprog -s mmi ~/bin/mmi.sh
```

This command will map file names of the form `/sfs/`*Host*`.mmi` to self-certifying pathnames stored in a user's `~/.ssh/known_hosts` directory, much as if that directory were a certification directory. If, however, no link of the appropriate name exists in `known_hosts`, then *mmi.sh* will attempt to fetch the pathname insecurely over the network and store it for subsequent accesses. The suffix `.mmi` stands for "man in the middle" as a reminder to users that such pathnames are not entirely secure.

Two SFS-specific commands are of note in *mmi.sh*. First, *dirsearch* with a `-l` flag to prints the contents of any symbolic link it finds, rather than the pathname of the link. (Removing the `-l` flag would not alter the end result of the script.) Second, *mmi.sh* uses the *sfskey hostid* command, which, given a host name or IP address, retrieves a self certifying pathname insecurely over the network and prints it to standard output.

7.4 Revocation paths

Section 3.5.3 suggested distributing self-authenticating revocation certificates through the file system itself. This can easily be implemented using *dirsearch* as an external revocation program. The one complication is that synchronous revocation checking of a server distributing revocation certificates can cause deadlock.

Figure 7.4 shows *revdir.sh*, a program that configures *revocation paths* analogous to certification paths. The program is invoked with a list of revocation directories, as *revdir.sh, dir1, [dir2, ...]*.

revdir.sh starts out by extracting the HostID from all remote directories in the revocation path. In the process, it ends up accessing each directory of the path, thereby running any other revocation programs on every element of the path. The script then passes the HostIDs to *sfskey norevokeset*, to disable any further synchronous revocation checking on HostIDs in the revocation path. Finally, *revdir.sh* adds *dirsearch* as a revocation program supplying the revocation path as its initial arguments. The `-c` flag to *dirsearch* causes it to print out the contents of any file it finds, rather than

```
#!/bin/sh

# Extract HostIDs from any remote directories in argument list
norev=
for dir in "$@"; do
    fullpath=`(cd $dir && /bin/pwd)` \
        && hostid=`expr "$fullpath" : "/sfs/[^/:]*:\([a-z0-9]*\)"` \
        && norev="$norev${norev:+ }$hostid"
done

# Disable synchronous revocation checking on those HostIDs
# to avoid deadlock
eval `echo sfskey norevokeset $norev`

# Search directory arguments for revocation certificates
test "$*" && sfskey revokeprog dirsearch -c "$@"
```

Figure 7-3: *revdir.sh*: Script to configure a revocation path

simply the path of the file. Thus, if revocation directories are populated with files containing revocation certificates, *dirsearch* will print those certificates to its standard output.

Chapter 8

Read-only file systems

Read-only data can have high performance, availability, and security needs. Some examples include executable binaries, popular software distributions, bindings from hostnames to public keys, and popular, static Web pages. In many cases, people widely replicate and cache such data to improve performance and availability—for instance, volunteers often set up mirrors of popular operating system distributions. Unfortunately, replication generally comes at the cost of security. Each replica adds a new opportunity for attackers to break in and tamper with data, or even for a replica's own administrator maliciously to serve modified data.

To address this problem, SFS supports a read-only dialect of the file system protocol that lets people widely replicate file systems on untrusted machines. We designed the read-only server software for high performance and scalability to many clients. Thus, it supports a wide range of applications for which one can't ordinarily use a network file system, including certification authorities like `/verisign` in Section 3.4.

Figure 8-1 shows the architecture of the read-only file system. The server-side software consists of two programs: *sfsrodb* and *sfsrosd*. Administrators run *sfsrodb* off-line, giving it the file system's private key and contents. *sfsrodb* converts the file system into a database and digitally signs it. Administrators then replicate the database on untrusted machines, where a simple, efficient, 400-line program, *sfsrosd*, serves the data to clients. The bulk of the read-only file system's functionality is implemented by clients, where a daemon *sfsrocd* actually verifies a database's contents

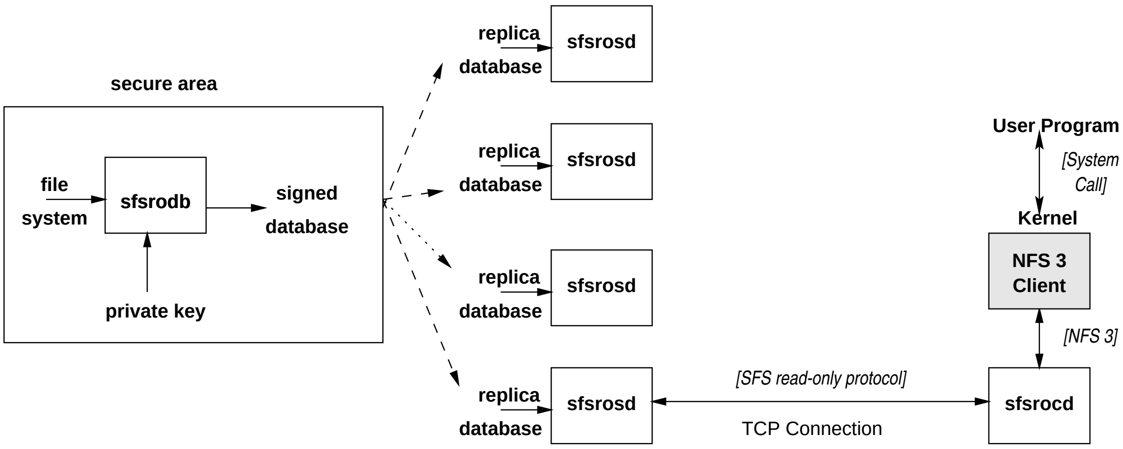


Figure 8-1: Architecture of the SFS read-only file system

```
struct sfsro1_signed_fsinfo {
  sfs_time start;
  unsigned duration;
  opaque iv[16];
  sfs_hash rootfh;
  sfs_hash fhdb;
};
```

Figure 8-2: Digitally signed root of an SFS read-only file system.

and interprets it as a file system.

The read-only protocol principally uses two RPCs: *getfsinfo*, and *getdata*. *getfsinfo* takes no arguments and returns a digitally signed `sfsro1_signed_fsinfo` structure, depicted in Figure 8-2. The SFS client verifies that this structure is signed by the private key matching the file system's *HostID*. The *getdata* RPC takes a 20-byte collision-resistant hash value as an argument and returns a data block producing that hash value. The client uses *getdata* to retrieve parts of the file system requested by the user, and verifies the authenticity of the blocks using the `sfsro1_signed_fsinfo` structure.

Because read-only file systems may reside on untrusted servers, the protocol relies on time to enforce consistency loosely but securely. The `start` field of `sfsro1_signed_fsinfo` indicates the time at which a file system was signed. Clients cache

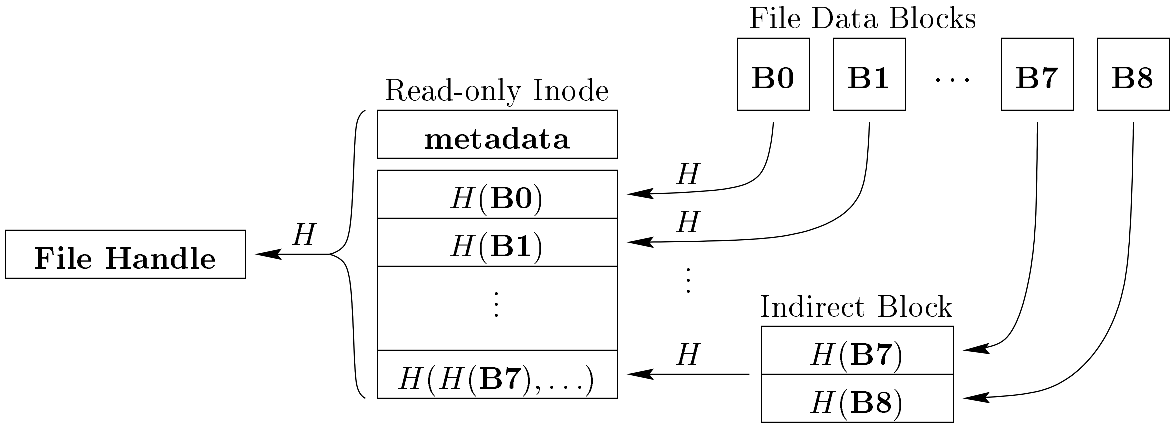


Figure 8-3: Format of a read-only file system inode.

the highest value they have seen, to prevent an attacker from rolling back the file system to a previous version. The `duration` field signifies the length of time for which the data structure should be considered valid. It represents a commitment on the part of a file system's owner to issue a newly signed file system within a certain period of time. Clients reject an `sfsro1_signed_fsinfo` structure when the current time exceeds `start` + `duration`.

The file system relies on a collision-resistant hash function, $H(x) = \text{SHA-1}(\texttt{iv}, x)$, to specify arbitrary-length blocks of data with a fixed size hash. `iv`, the initialization vector, is randomly chosen by *sfsrodb* the first time one creates a database for a file system. It ensures that simply knowing one particular collision of SHA-1 will not immediately give attackers collisions of functions actually used by SFS file systems.

`rootfh` is the file handle of the file system's root directory. It is a hash of the root directory's *inode* structure, which through recursive use of H specifies the contents of the entire file system. `fhdb` is the the hash of the root of a tree that contains every hash reachable from the root directory. `fhdb` lets clients securely verify that a particular hash is not in the database, so that they can return stale file handle errors when file systems change, as will be described later on.

Figure 8 shows the format of an inode in the read-only file system. The inode begins with some metadata, including the file's type (regular file, executable file, directory, or symbolic link), size, and modification time. It then contains hashes of successive 8K blocks of file data. If the file contains more than eight blocks, the inode contains the hash of an *indirect block*, which in turn contains hashes of file blocks. Similarly, for larger files, an inode can also contain the hash of double- and tripple-indirect blocks. In this way, the blocks of small files can be verified directly by the inode, while inodes can also indirectly verify large files—an approach similar to the on-disk data structures of the Berkeley Fast File System [19].

Each directory contains its full pathname from the root of the file system. Clients check this name when first looking up a directory, and use it to evaluate the file name "`..`" locally, using it as a reference for any directory's parent. The contents of a directory is simply a list of $\langle \text{name}, \text{file handle} \rangle$ pairs sorted lexicographically by name. Thus, clients can perform a binary search when looking up files in large directories, and avoid traversing the entire directory.

When a read-only file system is updated (by putting a new database on the server), a client may call *getdata* asking for the hash of a block no longer in the file system. When this occurs, the server returns an error, but the client cannot necessarily believe the error, as the server is not trusted. The client responds first by calling *getfsinfo* and checking that the `start` field has increased. This both ensures that the file system has been updated and gets the new root file handle. However, this is not sufficient for the client to return a stale file handle error to the user, as many files may persist across multiple versions of the file system.

To ensure that the file system database really does not contain a particular hash value, the client uses a B-tree maintained by the server whose root hashes to the `fhdb` field of the `sfsro1_signed_fsinfo` structure. The tree's structure is similar to the indirect blocks used by inode structures. Interior nodes contain hashes of child blocks, while the leaf nodes contain all hashes in the file system in sorted order. By fetching blocks from the tree (also using *getdata*), the client can prove to itself that a particular file or block really has disappeared from the file system, and that it should return a stale file handle error to the user.

Chapter 9

Performance

In designing SFS we ranked security, extensibility, and portability over performance. Our performance goal was modest: to make application performance on SFS comparable to that on NFS, a widely used network file system. This section presents results that show that SFS does slightly worse than NFS 3 over UDP and better than NFS 3 over TCP.

9.1 Experimental setup

We measured file system performance between two 550 MHz Pentium IIIs running FreeBSD 3.3. The client and server were connected by 100 Mbit/sec switched Ethernet. Each machine had a 100 Mbit SMC EtherPower Ethernet card, 256 Mbytes of memory, and an IBM 18ES 9 Gigabyte SCSI disk. We report the average of multiple runs of each experiment.

To evaluate SFS's performance, we ran experiments on the local file system, NFS 3 over UDP, and NFS 3 over TCP. SFS clients and servers communicate with TCP, making NFS 3 over TCP the ideal comparison to isolate SFS's inherent performance characteristics. Since NFS requires IP fragmentation with UDP, TCP is also the preferred trasport for NFS traffic over anything but a local network. We decided to concentrate on the comparison with NFS over UDP, however, as many people still use UDP for NFS. Consequently, FreeBSD's TCP NFS code may be less well optimized.

File System	Latency (μsec)	Throughput (Mbyte/sec)
NFS 3 (UDP)	200	9.3
NFS 3 (TCP)	220	7.6
SFS	790	4.1
SFS w/o encryption	770	7.1

Figure 9-1: Micro-benchmarks for basic operations.

SFS itself uses UDP for NFS traffic it exchanges with the local operating system, and so is unaffected by any limitations of FreeBSD's TCP NFS.

9.2 SFS base performance

Three principal factors make SFS's performance different from NFS's. First, SFS has a user-level implementation while NFS runs in the kernel. This hurts both file system throughput and the latency of file system operations. Second, SFS encrypts and MACs network traffic, reducing file system throughput. Finally, SFS has better attribute and access caching than NFS, which reduces the number of RPC calls that actually need to go over the network.

To characterize the impact of a user-level implementation and encryption on latency, we measured the cost of a file system operation that always requires a remote RPC but never requires a disk access—an unauthorized *fchown* system call. The results are shown in the Latency column of Figure 9-1. SFS is 4 times slower than both TCP and UDP NFS. Only 20 μsec of the 590 μsec difference can be attributed to software encryption; the rest is the cost of SFS's user-level implementation.

To determine the cost of software encryption, we measured the speed of streaming data from the server without going to disk. We sequentially read a sparse, 1,000 Mbyte file. The results are shown in the Throughput column of Figure 9-1. SFS pays 3 Mbyte/sec for its user-level implementation and use of TCP, and a further 2.2 Mbyte/sec for encryption.

Although SFS pays a substantial cost for its user-level implementation and soft-

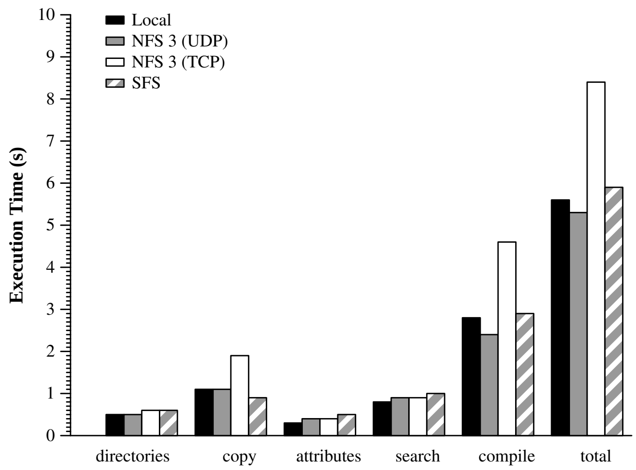


Figure 9-2: Wall clock execution time (in seconds) for the different phases of the modified Andrew benchmark, run on different file systems. Local is FreeBSD's local FFS file system on the server.

ware encryption in these benchmarks, several factors mitigate the effects on application workloads. First, multiple outstanding request can overlap the latency of NFS RPCs. Second, few applications ever read or write data at rates approaching SFS's maximum throughput. Disk seeks push throughput below 1 Mbyte/sec on anything but sequential accesses. Thus, the real effect of SFS's encryption on performance is to increase CPU utilization rather than to cap file system throughput. Finally SFS's enhanced caching improves performance by reducing the number of RPCs that need to travel over the network.

9.3 End-to-end performance

We evaluate SFS's application performance with the Modified Andrew Benchmark (MAB) [20]. The first phase of MAB creates a few directories. The second stresses data movement and metadata updates as a number of small files are copied. The third phase collects the file attributes for a large set of files. The fourth phase searches the files for a string which does not appear, and the final phase runs a compile. Although MAB is a light workload for today's file systems, it is still relevant, as we are more interested in protocol performance than disk performance.

System	Time (seconds)
Local	140
NFS 3 (UDP)	178
NFS 3 (TCP)	207
SFS	197

Figure 9-3: Compiling the GENERIC FreeBSD 3.3 kernel.

Figure 9-2 shows the execution time of each MAB phase and the total. As expected, the local file system outperforms network file systems on most phases; the local file system performs no network communication and does not flush data to disk on file closes. The local file system is slightly slower on the compile phase because the client and server have a larger combined cache than the server alone.

Considering the total time for the networked file systems, SFS is only 11% (0.6 seconds) slower than NFS 3 over UDP. SFS performs reasonably because of its more aggressive attribute and access caching. Without enhanced caching, MAB takes a total of 6.6 seconds, 0.7 seconds slower than with caching and 1.3 seconds slower than NFS 3 over UDP.

We attribute most of SFS's slowdown on MAB to its user-level implementation. We disabled encryption in SFS and observed only an 0.2 second performance improvement.

To evaluate how SFS performs on a larger application benchmark, we compiled the GENERIC FreeBSD 3.3 kernel. The results are shown in Figure 9-3. SFS performs 16% worse (29 seconds) than NFS 3 over UDP and 5% better (10 seconds) than NFS 3 over TCP. Disabling software encryption in SFS sped up the compile by only 3 seconds or 1.5%.

9.4 Sprite LFS microbenchmarks

The small file test of the Sprite LFS microbenchmarks [25] creates, reads, and unlinks 1,000 1 Kbyte files. The large file test writes a large (40,000 Kbyte) file sequentially, reads from it sequentially, then writes it randomly, reads it randomly, and finally

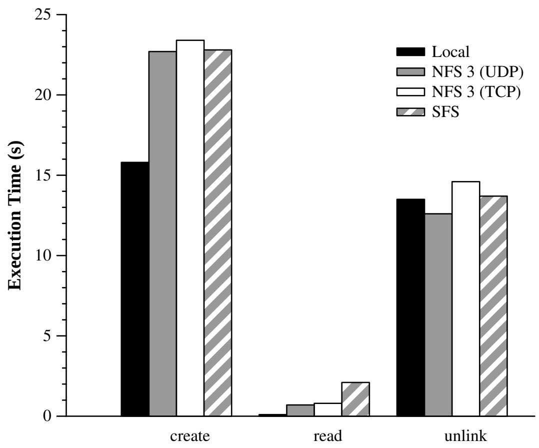


Figure 9-4: Wall clock execution time for the different phases of the Sprite LFS small file benchmark, run over different file systems. The benchmark creates, reads, and unlinks 1,000 1 Kbyte files. Local is FreeBSD's local FFS file system on the server.

reads it sequentially. Data is flushed to disk at the end of each write phase.

The small file benchmark operates on small files, does not achieve high disk throughput on FreeBSD's FFS file system, and therefore mostly stresses SFS's latency. On the create phase, SFS performs about the same as NFS 3 over UDP (see Figure 9-4). SFS's attribute caching makes up for its greater latency in this phase; without attribute caching SFS performs 1 second worse than NFS 3. On the read phase, SFS is 3 times slower than NFS 3 over UDP. Here SFS suffers from its increased latency. The unlink phase is almost completely dominated by synchronous writes to the disk. The RPC overhead is small compared to disk accesses and therefore all file systems have roughly the same performance.

The large file benchmark stresses throughput and shows the impact of both SFS's user-level implementation and software encryption. On the sequential write phase, SFS is 4.4 seconds (44%) slower than NFS 3 over UDP. On the sequential read phase, it is 5.1 seconds (145%) slower. Without encryption, SFS is only 1.7 seconds slower (17%) on sequential writes and 1.1 seconds slower (31%) on sequential reads.

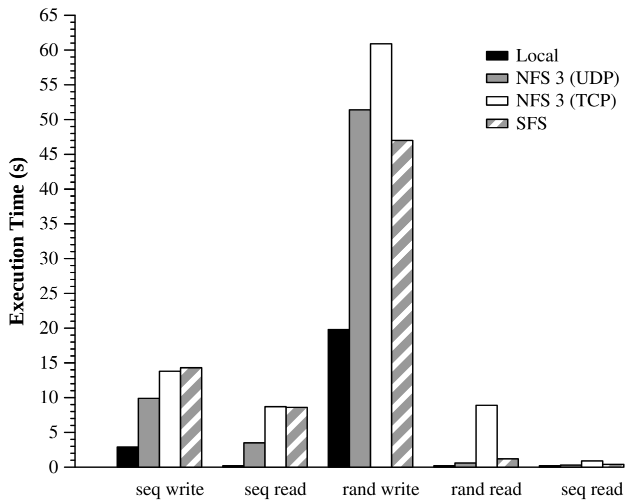


Figure 9-5: Wall clock execution time for the different phases of the Sprite LFS large file benchmarks, run over different file systems. The benchmark creates a 40,000 Kbyte file and reads and writes 8 Kbyte chunks. Local is FreeBSD's local FFS file system on the server.

Operation	msec
Encrypt	1.11
Decrypt	39.62
Sign	40.56
Verify	0.10

Automounter	msec	
SFS mount	64	
SFS auth	49	
SFS both	109	
NFS amd	10–1,000	(unfair)

Figure 9-6: Cost of public key operations and SFS session protocols. All measurements were performed using 1,280-bit Rabin keys, the default in SFS.

9.5 Public key protocols

One performance characteristic of SFS not captured by traditional file system benchmarks is the cost associated with public key cryptography. SFS's key negotiation protocol from Section 4.1 requires two public key encryptions and two decryptions. The user authentication protocol in Section 4.2 requires a digital signature and verification. These operations add latency to the very first file access a user makes to a particular file system.

The left hand table in Figure 9.5 shows the cost of various public key operations using 1,280-bit Rabin-Williams keys (the default in SFS). Decryption and signing each require a 1,280-bit modular exponentiation (which SFS performs as two 640-bit modular exponentiations using the Chinese remainder theorem), pushing the cost around 40 milliseconds. Encryption and signature verification, on the other hand, do not require modular exponentiation and are consequently much faster.

The right hand table in Figure 9.5 shows the cost of various operations on SFS. (The numbers reported are the average of 5 experiments, all of which had virtually identical results.) "SFS mount" is the time to access an unmounted file system for the first time, anonymously, when a user is not running an agent. This includes the cost of setting up a connection, performing the key negotiation protocol, and mounting the file system on the client. The measured time, 64 milliseconds, compares favorably to the roughly 80 millisecond computational cost of the key negotiation protocol. This is because one public key decryption occurs on the client and one on the server. SFS performs both concurrently, overlapping computation time to reduce protocol latency.

Also in the right hand table, "SFS auth" reports the time it takes to authenticate a user to the server of an already mounted file system. This includes the cost of the user authentication protocol, two calls into the user's agent (one for authentication, one for a null revocation check), and one call into *sfsauthd* on the server side. "SFS both" measures the time of mounting a file system and authenticating a user for the same file access. SFS saves a small amount of time because it only needs to delay one file access while performing the mount and revocation checks. (Section 5.3 describes how SFS delays file accesses by redirecting them to "`.mnt`" file systems with symbolic links).

To determine the significance of these measurements, we also measured the NFS automounter *amd*. We could not perform a fair comparison, because *amd* and SFS's automounter offer very different functionality. In fact, *amd*'s performance varies widely depending on its configuration. We looked at a number of *amd* configurations in different administrative domains, and measured mount times of anywhere from 10 millisecond to a whole second. From this we conclude that people could greatly optimize *amd*'s performance by tuning its configuration, but that most people probably do not care. People care about application performance on file systems, but are apparently willing to wait a second for the first file access. Thus, the latency of SFS's

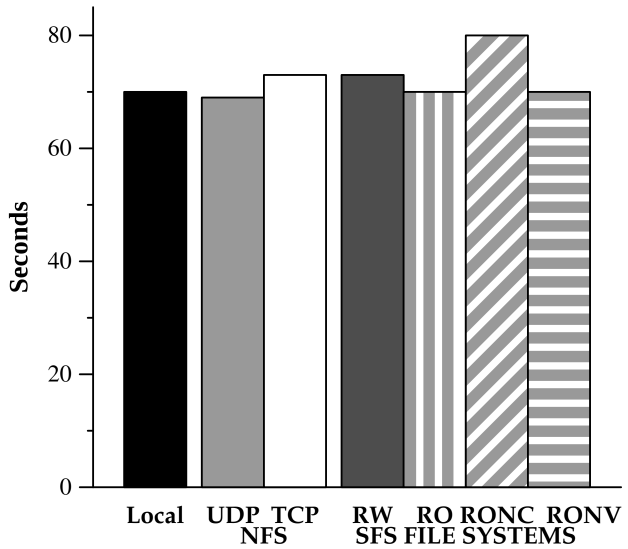


Figure 9-7: Compiling the Emacs 20.6 source.

protocols should not be noticeable.

9.6 Read-only file system performance

Though SFS's read-write dialect offers acceptable latency for mounting file systems, each connection requires a fair amount of computation from the server—40 milliseconds on a 550 MHz Pentium III. This cost limits the number of connections per second a server can accept. The fact that servers need a file system's private key for each connection also restricts the ability to replicate servers. SFS's read-only dialect eliminates these problems by freeing servers from the need to keep any private keys or perform any cryptographic operations on-line. We demonstrate the read-only server's scalability by measuring two applications: software distribution and a certification authority.

9.6.1 Software distribution

To evaluate how well the read-only file system performss on software distribution, we measured the time compiled to compile (without optimization or debugging) Emacs

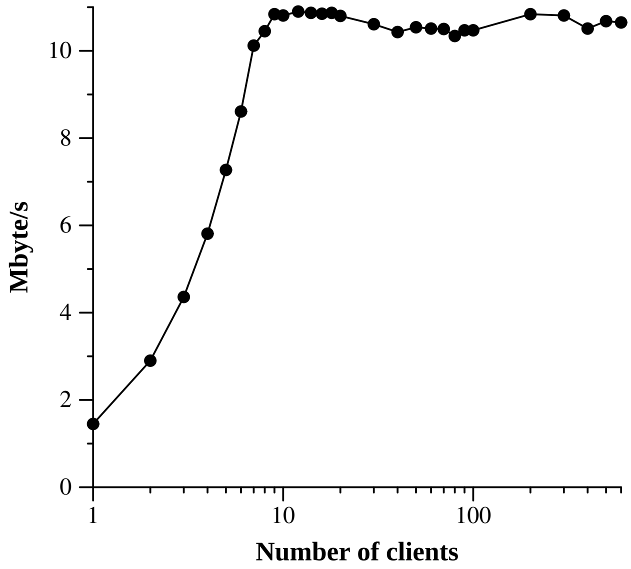


Figure 9-8: The throughput delivered by the server for increasing number of simultaneous clients that compile the Emacs 20.6 source. The number of clients is plotted on a log scale.

20.6 from a read-only distribution server in a local build directory. The results are shown in Figure 9-7. SFSRO performs 1% worse (1 second) than NFS 3 over UDP and 4% better (3 seconds) than NFS 3 over TCP. Disabling integrity checks in the read-only file system (RONV) did not speed up the compile because the client's caches absorb the cost of hash verification. However, disabling caching did decrease performance (see the bar for RONC).

To evaluate how well `sfsrosd` scales, we took a trace of a single client compiling the Emacs and repeatedly played the trace to the server from an increasing number of simulated, concurrent clients. We measured the total aggregate throughput of file system data from the server to all clients (not counting RPC headers). Figure 9-8 shows the results. `sfsrosd` scales linearly until it hits 10 clients, at which point it reaches the maximum bandwidth of the 100 Mbit/s Ethernet. When we increase the number of clients further, the server sustains a data rate of over 10 Mbytes/sec until 600 concurrent clients. (At 700, the server's FreeBSD kernel reliably panics—a bug not attributable to SFS's user-level server implementation.)

9.6.2 Certificate authority

To measure how well SFSRO does under workloads with much shorter-lived connections than an emacs compile, we measured the number of clients that could connect to an SFSRO server and read one symbolic link. This would be the expected workload for a machine serving as a certification authority like `/verisign`. For comparison, we compare SFSRO's performance to the number of connections per second sustainable by the SFS read-write server, by a secure Apache SSL server, and by an ordinary, insecure Apache HTTP server.

The SFS read-write server performs one Rabin-Williams decryption per connection (using a 1,024-bit key for fair comparison with SSL), while the SFS read-only server performs no on-line cryptographic operations. The Web server is Apache 1.3.12 with OpenSSL 0.9.5a and ModSSL 2.6.3-1.3.12. Our SSL ServerID certificate and Verisign CA certificate use 1,024-bit RSA keys. All the SSL connections use the TLSv1 cipher suite consisting of Ephermal Diffie-Hellman key exchange, DES-CBC3 for confidentiality, and SHA-1 HMAC for integrity.

To generate enough load to saturate the server, we wrote a simple client program that sets up connections, reads a symbplic link containing a certificate, and terminates the connection as fast as it can. We ran this client program on multiple machines. In all experiments, the certificate was in the main-memory of the server, so the numbers reflect software performance, not disk performance. This scenario is realistic, since we envision that important on-line certificate authorities would have large memories to avoid disk accesses, like DNS root servers.

Figure 9-9 shows that the read-only server scales well. The SFS read-only server can process 26 times more connections than the SFS read-write server because the read-only server performs no on-line cryptographic operations. The read-write server is bottlenecked by private key decryptions, which take 24 msec for 1,024-bit keys. Hence, the read-write server can at best achieve 40 (1000/24) connections per second. By comparing the read-write server to the Apache Web server with SSL, we see that the read-write server is in fact quite efficient; the SSL protocol requires more

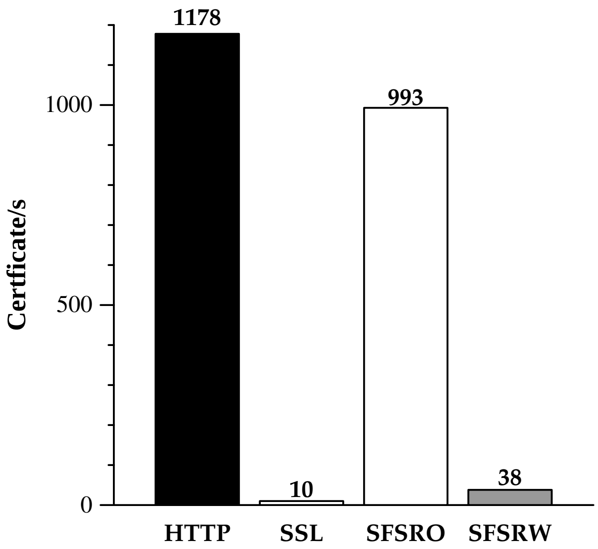


Figure 9-9: Maximum sustained certificate downloads per second for different systems. HTTP is an insecure Web server, SSL is a secure Web server, SFSRW is the secure SFS read-write file system, and SFSRO is the secure read-only file system.

cryptographic operations on the server than the SFS protocol.

By comparing the read-only server with an insecure Apache server, we can conclude that the read-only server is a good platform for serving read-only data to many clients; the number of connections per second is only 16% lower than that of an insecure Apache server. In fact, its performance is within an order of magnitude of the performance of a DNS root server, which according to Network Solutions can sustain about 4,000 lookups per second (DNS uses UDP instead of TCP). Since the DNS root servers can support on-line name resolution for the Internet, this comparison suggests that it is perfectly reasonable to built a distributed on-line certificate authority using SFS read-only servers.

9.7 Summary

The experiments demonstrate that SFS's user-level implementation and software encryption carry a performance price. Nonetheless, SFS can achieve acceptable performance on application workloads, in part because of its better caching than NFS 3.

Moreover, SFS's read-only file system has performance rivaling that of insecure web servers. Thus, read-only file systems can realistically support the requirements of on-line certification authorities.

We expect SFS's performance penalty to decline as hardware improves. The relative performance difference of SFS and NFS 3 on MAB shrunk by a factor of two when we moved from 200 MHz Pentium Pros to 550 MHz Pentium IIIs; this trend should continue.

Chapter 10

Conclusions

The challenge in achieving security at a global scale is key management. Cryptography has become computationally inexpensive enough that one can now encrypt and MAC most network file system traffic. In a system as large and decentralized as the Internet, however, there cannot exist a global consensus on how to manage encryption keys.

While some hold out hope for a global Public Key Infrastructure (PKI), no authority will ever be trusted to manage public keys for every server on the Internet or dictate the terms of setting up new servers. How can unclassified military networks in many countries all trust the same certification authority? Even given a universally trusted authority, no certification procedure can provide the right level of rigor for all servers. At one extreme, the process of certifying public keys will be prohibitively complex and costly, hindering the creation of new servers. At the other extreme, attackers will so easily obtain fraudulent certificates that the system's security will not satisfy most needs. No single procedure can satisfy users who range from the military to on-line merchants to students setting up servers in dormitory rooms.

Perhaps more importantly, even in a global system, key management is sometimes a purely local issue. For instance, a user might quite reasonably wish to access the server at which he physically typed his password when creating an account. Such a security requirement cannot be expressed to third parties; no one can verify the server's identity without knowing a secret password. No global key management can help users fulfill local security requirements. Given that passwords are the single most prevalent authentication mechanism in use today, one cannot hope to replace them with a PKI that provides neither the same convenience nor the same security properties.

SFS shows that one can achieve global security precisely by not trying to perform global key management. Previous secure applications have not scaled to the size of the Internet because they have based their security on one particular form of key management. For instance, a secure web browser takes a URL as input, obtains and certifies an SSL public key for the URL's hostname, uses the public key to authenticate a secure channel to the server, and then retrieves and displays a document. SSL key management is inextricably linked to the browser's other functions; one cannot retrieve and display a document without appropriate certificates, even if one already knows the server's public key.

Applications can be all the more useful if they provide security independent of any key management. SFS performs no internal key management. Of course, one needs key management to use SFS, but the file system's security guarantee—that file contents match HostIDs—does not depend on the user's choice of key management. Thus, anyone developing a new key management scheme can immediately apply it to SFS—obtaining a file system based on the new scheme without actually needing to implement a file system.

Conversely, key management is easiest to implement if one can harness secure applications as building blocks. SFS constitutes a particularly effective building block for key management because it essentially embeds public keys in the file system while simultaneously providing a secure file system. Thus, key management is never finished or out of context; any secure file server can provide key management—either directly with symbolic links, or indirectly by serving data to certification programs launched from user agents.

The complications of key management have prevented any previous secure file system from gaining widespread use. SFS solves the problem by showing that secure file systems need not in fact perform key management. Self-certifying pathnames specify public keys in file names, making the file system's security independent of key management. Symbolic links assign human-readable names to self-certifying pathnames, letting users exploit the secure file system itself for key management. Finally, agents allow users to plug arbitrary key management algorithms into the file system.

We hope SFS will let many people enjoy secure file sharing over the Internet, while also sparing them the all too common inconvenience of facing inappropriate key management. To facilitate its deployment, SFS is free software.

Appendix A

SFS Protocols

This appendix gives down-to-the-byte descriptions of the SFS protocols previously described at a higher level. We describe the protocols in the XDR protocol description language [31], which completely describes the exact bytes composing RPC messages. First, however, we must describe SFS's encoding of big integers and the exact algorithms it uses for public key cryptography. We also describe two extensions SFS needed to make to Sun RPC to support session encryption and passing of file descriptors between processes.

A.1 `bigint` encoding

In addition to the standard XDR datatypes, SFS's RPC messages contain data elements of the type `bigint`. `bigint`s are encoded as variable length opaque data. That is, they can be correctly marshalled and unmarshalled with the definition:

```
typedef opaque bigint<>;
```

Inside this XDR-defined opaque vector of bytes, SFS encodes `bigint`s as two's complement numbers in big-endian order. Because the SFS protocols only involve positive numbers, the only visible consequence the two's complement encoding is that one must leave space for a sign bit in the most significant bit of each encoded number. Thus, for instance, the hexadecimal number 0x8642 actually requires three bytes to encode as {`0x00, 0x86, 0x42`}.

A.2 Public key functions

SFS public keys are large integers n, such that $n = pq$, where $p = 3 \bmod 8$ and $q = 7 \bmod 8$. p and q constitute the private key. SFS uses the following four functions suggested by Williams [34], where J is the Jacobi function:

$$E1(m) = \begin{cases} 8m+4 & \text{if } J(2m+1,n) = 1 \\ 4m+2 & \text{if } J(2m+1,n) = -1 \end{cases}$$

$$E2(m) = m^2 \bmod n$$

$$D2(m) = m^{((p-1)(q-1)+4)/8} \bmod n$$

$$D1(m) = \begin{cases} (m-4)/8 & \text{if } m \equiv 0 \pmod 4 \\ (n-m-4)/8 & \text{if } m \equiv 1 \pmod 4 \\ (m-2)/4 & \text{if } m \equiv 2 \pmod 4 \\ (n-m-2)/4 & \text{if } m \equiv 3 \pmod 4 \end{cases}$$

In addition, SFS defines a family of functions it calls "SHA-1 oracles," inspired by [1]. The set of SHA-1 oracles is indexed by 64-bit integers. Letting $\langle k \rangle$ represent the 8-byte, big-endian encoding of the number k, and letting $\|$ designate concatenation, the SHA-1 oracle S_i is defined to output the following series of bytes:

$$S_i(m) = \text{SHA-1}\left(\langle 0 \rangle \| \langle i \rangle \| m\right) \;\|\; \text{SHA-1}\left(\langle 1 \rangle \| \langle i \rangle \| m\right) \;\|\; \text{SHA-1}\left(\langle 2 \rangle \| \langle i \rangle \| m\right) \;\|\; \cdots$$

$S_i/l(m)$ designates the first l bits of this sequence.

With this scheme, an an encryption of a message m is computed as $E2(E1(\textit{pre-encrypt}(m)))$, using the function defined in Figure A.2. A ciphertext c is decrypted by inverting the *pre-encrypt* function and running it on the output of $D1(D2(c))$. Note that *pre-encrypt* cannot be inverted on bit strings if an attacker supplies a ciphertext that was not legitimately produced using *pre-encrypt*. In such cases, SFS behaves as

bigint
pre-encrypt (bigint $\mathbf{n}$, string $\mathbf{m}$)
{
  string $\mathbf{Mz} \leftarrow \mathbf{m} \,\|\, 0^{128}$;
  int $\mathbf{padsize} \leftarrow |\mathbf{n}| - 5 - |\mathbf{Mz}|$;
  string $\mathbf{r} \leftarrow$ ($\mathbf{padsize}$ random bits) $\|\ 0^{-\mathbf{padsize}\,\&\,7}$;   [0-padded to even number of bytes]
  string $\mathbf{Mg} \leftarrow \mathbf{Mz} \oplus S_1/|\mathbf{Mz}|(\mathbf{r})$;
  string $\mathbf{Mh} \leftarrow \mathbf{r} \oplus S_2/|\mathbf{r}|(\mathbf{Mg})$;
  return $\mathbf{Mg} \,\|\, \mathbf{Mh}$ interpreted as a little-endian bigint;
}

Figure A-1: Definition of *pre-encrypt*, a "plaintext-aware" instantiation of OAEP [2]. $|x|$ designates the size in bits of the number or string x. 0^l designates l 0-valued bits.

though the decryption were a random string.

To sign a message m, SFS computes $s = D2(E1(\textit{pre-sign}(m)))$, using the function in Figure A.2, and outputs this as a signature[1]. Verification of signature s takes place by first computing $D1(E2(s))$, then checking that that number really was produced by the *pre-sign*. As part of user authentication, the `sfs_signed_authreq` structure is actually signed recoverably, using the function *pre-sign-recoverable* in Figure A.2 rather than *pre-sign*. This allows *sfsauthd* to extract the structure from the signature itself.

A.3 Encrypting transport

SFS encrypts and MACs RPC traffic at the RPC transport layer. Encryption is performed by ARC4 [14], with a slight modification. During the key schedule, the key is broken up into 16-byte segments, and the key schedule is performed once on each segment. The MAC is SHA-1-based. It is non-standard, because SFS takes advantage of knowing the packet length ahead of time and knowing that the MAC value will be

[1] Actually, SFS chooses two random bits, b_p and b_q, and outputs as a signature the number $s' \in \mathbf{Z}_n^*$ such that $s' \equiv (-1)^{b_p} s \pmod{p}$ and $s' \equiv (-1)^{b_q} s \pmod{q}$.

bigint
pre-sign (bigint $\mathbf{n}$, string $\mathbf{m}$)
{
  int $\mathbf{padsize} \leftarrow |\mathbf{n}| - 165$;
  string $\mathbf{r} \leftarrow 128$ random bits;
  string $\mathbf{M1} \leftarrow \text{SHA-1}(\mathbf{m} \,\|\, \mathbf{r})$;
  string $\mathbf{r1} \leftarrow \mathbf{r} \,\|\, 0^{\mathbf{padsize}-128}$;
  string $\mathbf{Mg} \leftarrow \mathbf{r1} \oplus S_3/\mathbf{padsize}(\mathbf{M1})$;
  return $\mathbf{M1} \,\|\, \mathbf{Mg}$ interpreted as a little-endian bigint;
}

Figure A-2: Definition of *pre-sign*, an instantiation of the approach described in [3].

bigint
pre-sign-recoverable (bigint $\mathbf{n}$, string $\mathbf{m}$)
{
  int $\mathbf{padsize} \leftarrow |\mathbf{n}| - 165$;
  string $\mathbf{r} \leftarrow 128$ random bits;
  string $\mathbf{M1} \leftarrow \text{SHA-1}(\mathbf{m} \,\|\, \mathbf{r})$;
  string $\mathbf{r1} \leftarrow \mathbf{r} \,\|\, \mathbf{m} \,\|\, 0^{\mathbf{padsize}-|\mathbf{m}|-128}$;
  string $\mathbf{Mg} \leftarrow \mathbf{r1} \oplus S_4/\mathbf{padsize}(\mathbf{M1})$;
  return $\mathbf{M1} \,\|\, \mathbf{Mg}$ interpreted as a little-endian bigint;
}

Figure A-3: Definition of *pre-sign-recoverable*, an instantiation of the approach described in [3] which allows recovery of messages (of known length) from signatures.

encrypted. Thus, SFS's MAC is tightly coupled with the ARC4 encryption.

The encrypting transport uses two distinct encryption keys, one for each direction. Each key is used to initialize an ARC4 stream cipher in SFS's non-standard way. To encrypt and MAC a packet of length n (where n is divisible by 4), the transport performs the following calculations:

- Let $A[0]$ be the next byte of the ARC4 stream, $A[1]$ following, etc.
- Let $M[0] \dots M[n-1]$ be the message to encrypt
- Let $R[0] \dots R[n+19]$ be the encryption result transmitted on the wire
- Let SHA-1/16 designate the first 16 bytes of a SHA-1 hash
- Let $P[0] \dots P[3] \leftarrow \texttt{ntonl}(0\text{x}80000000|n)$
- Let $P[4] \dots P[3+n] \leftarrow M[0] \dots M[n-1]$
- Let $P[4{+}n] \dots P[4{+}n{+}15] \leftarrow \text{SHA-1/16}(A[0] \dots A[15] \parallel P[0] \dots p[3{+}n] \parallel A[16] \dots A[31])$
- for $(i \leftarrow 0;\ i \leq n+19;\ i \leftarrow i+1)\ \{R[i] \leftarrow P[i] \oplus A[32+i]\}$

In other words, the first 32 bytes of ARC4 data get used to compute a 16-byte MAC on the message length and contents. Then then entire packet including length, message contents, and MAC are encrypted by XORing them with subsequent bytes from the ARC4 stream.

In the RPC procedure `SFSPROC_ENCRYPT`, the server replies, in cleartext, before performing its decryption. This allows the client and server to overlap their decryption computations, and reduces the latency of the protocol. If any RPCs are pending at the time a server receives an `SFSPROC_ENCRYPT` message, the server aborts the connection. (Otherwise, an attacker could inject an unMACed request before the encryption started, and the reply would come over the encrypted, MACed channel, potentially confusing the client.)

A.4 File descriptor passing

SFS makes extensive use of file descriptor passing between processes over Unix domain sockets. SFS makes a simple extension to the Sun RPC TCP transport specification in [30] to define a Unix domain transport over which file descriptors can be passed. Over TCP connections, Sun RPC delimits messages by prefixing lengths to message fragments. Each message fragment is prefixed by a 1-bit end of message marker and a 31-bit length in big-endian order. Message fragments are accumulated until one has the end of message marker set, at which point they are concatenated and processed.

There is no reason ever to transmit a zero-length message fragment that is not the end of a message. Thus, SFS reserves the fragment prefix of four zero bytes as a transport-specific marker alerting the remote process that a file descriptor will be passed over a Unix-domain socket. To pass a file descriptor over a Unix domain transport, then, SFS transmits four zero bytes, then sends the file descriptor. Because some variants of Unix require a file descriptor to be passed along with data, SFS also writes the byte 125 as it is sending a file descriptor.

A.5 Session Protocols

```
/* $Id: sfs_prot.x,v 1.58 2000/04/11 22:05:54 fubob Exp $ */

/*
 * This file was written by David Mazieres.  Its contents is
 * uncopyrighted and in the public domain.  Of course, standards of
 * academic honesty nonetheless prevent anyone in research from
 * falsely claiming credit for this work.
 */

%#include "bigint.h"

#ifdef SFSSVC
%#include "nfs3exp_prot.h"
#else /* !SFSSVC */
%#include "nfs3_prot.h"
const ex_NFS_PROGRAM = 344444;
const ex_NFS_V3 = 3;
#endif

#ifdef RPCC
```

```
# ifndef UNION_ONLY_DEFAULT
#  define UNION_ONLY_DEFAULT 1
# endif /* UNION_ONLY_DEFAULT */
#endif

#ifndef FSINFO
#define FSINFO sfs_fsinfo
#endif /* !FSINFO */

const SFS_PORT = 4;
const SFS_RELEASE = 4;          /* 100 * release no. where protocol changed */

enum sfsstat {
  SFS_OK = 0,
  SFS_BADLOGIN = 1,
  SFS_NOSUCHHOST = 2,
  SFS_NOTSUPP = 10004,
  SFS_TEMPERR = 10008,
  SFS_REDIRECT = 10020
};

/* Types of hashed or signed messages */
enum sfs_msgtype {
  SFS_HOSTINFO = 1,
  SFS_KSC = 2,
  SFS_KCS = 3,
  SFS_SESSINFO = 4,
  SFS_AUTHINFO = 5,
  SFS_SIGNED_AUTHREQ = 6,
  SFS_AUTHREGISTER = 7,
  SFS_AUTHUPDATE = 8,
  SFS_PATHREVOKE = 9,
  SFS_KEYCERT = 10,
  SFS_ROFSINFO = 11
};

/* Type of service requested by clients */
enum sfs_service {
  SFS_SFS = 1,                  /* File system service */
  SFS_AUTHSERV = 2,             /* Crypto key server */
  SFS_REX = 3                   /* Remote execution */
};

typedef string sfs_extension<>;
typedef string sfs_hostname<222>;
typedef opaque sfs_hash[20];
typedef opaque sfs_secret[16];
typedef unsigned hyper sfs_seqno;
typedef unsigned hyper sfs_time;

typedef bigint sfs_pubkey;
typedef bigint sfs_ctext;
typedef bigint sfs_sig;
```

```
struct sfs_hashcharge {
  unsigned int bitcost;
  sfs_hash target;
};
typedef opaque sfs_hashpay[64];


/*
 * Hashed structures
 */

/* Two, identical copies of of the sfs_hostinfo structure are
 * concatenated and then hashed with SHA-1 to form the hostid. */
struct sfs_hostinfo {
  sfs_msgtype type;             /* = SFS_HOSTINFO */
  sfs_hostname hostname;
  sfs_pubkey pubkey;
};

struct sfs_connectinfo {
  unsigned release;             /* Client release */
  sfs_service service;
  sfs_hostname name;            /* Server hostname */
  sfs_hash hostid;              /* = SHA1 (sfs_hostinfo, sfs_hostinfo) */
  sfs_extension extensions<>;
};

struct sfs_servinfo {
  unsigned release;             /* Server release */
  sfs_hostinfo host;            /* Server hostinfo */
  unsigned prog;
  unsigned vers;
};

/* The two shared session keys, ksc and kcs, are the SHA-1 hashes of
 * sfs_sesskeydat with type = SFS_KCS or SFS_KSC.  */
struct sfs_sesskeydat {
  sfs_msgtype type;             /* = SFS_KSC or SFS_KCS */
  sfs_servinfo si;
  sfs_secret sshare;            /* Server's share of session key */
  sfs_connectinfo ci;
  sfs_pubkey kc;
  sfs_secret cshare;            /* Client's share of session key */
};

/* The sessinfo structure is hashed to produce a session ID--a
 * structure both the client and server know to be fresh, but which,
 * unlike the session keys, can safely be divulged to 3rd parties
 * during user authentication.  */
struct sfs_sessinfo {
  sfs_msgtype type;             /* = SFS_SESSINFO */
  opaque ksc<>;                 /* = SHA-1 ({SFS_KSC, ...}) */
  opaque kcs<>;                 /* = SHA-1 ({SFS_KCS, ...}) */
};
```

```
/* The authinfo structure is hashed to produce an authentication ID.
 * The authentication ID can be computed by an untrusted party (such
 * as a user's unprivileged authentication agent), but allows that
 * third party to verify or log the hostname and hostid to which
 * authentication is taking place.  */
struct sfs_authinfo {
  sfs_msgtype type;             /* = SFS_AUTHINFO */
  sfs_service service;
  sfs_hostname name;
  sfs_hash hostid;              /* = SHA-1 (sfs_hostinfo, sfs_hostinfo) */
  sfs_hash sessid;              /* = SHA-1 (sfs_sessinfo) */
};

/*
 * Public key ciphertexts
 */

struct sfs_kmsg {
  sfs_secret kcs_share;
  sfs_secret ksc_share;
};

/*
 * Signed messages
 */

struct sfs_keycert_msg {
  sfs_msgtype type;             /* = SFS_KEYCERT */
  unsigned duration;            /* Lifetime of certificate */
  sfs_time start;               /* Time of signature */
  sfs_pubkey key;               /* Temporary public key */
};

struct sfs_keycert {
  sfs_keycert_msg msg;
  sfs_sig sig;
};

struct sfs_signed_authreq {
  sfs_msgtype type;             /* = SFS_SIGNED_AUTHREQ */
  sfs_hash authid;              /* SHA-1 (sfs_authinfo) */
  sfs_seqno seqno;              /* Counter, value unique per authid */
  opaque usrinfo[16];           /* All 0s, or <= 15 character logname */
};

struct sfs_redirect {
  sfs_time serial;
  sfs_time expire;
  sfs_hostinfo hostinfo;
};

/* Note: an sfs_signed_pathrevoke with a NULL redirect (i.e. a
 * revocation certificate) always takes precedence over one with a
```

```
 * non-NULL redirect (a forwarding pointer). */
struct sfs_pathrevoke_msg {
  sfs_msgtype type;             /* = SFS_PATHREVOKE */
  sfs_hostinfo path;            /* Hostinfo of old self-certifying pathname */
  sfs_redirect *redirect;       /* Optional forwarding pointer */
};

struct sfs_pathrevoke {
  sfs_pathrevoke_msg msg;
  sfs_sig sig;
};

/*
 * RPC arguments and results
 */

typedef sfs_connectinfo sfs_connectarg;

struct sfs_connectok {
  sfs_servinfo servinfo;
  sfs_hashcharge charge;
};

union sfs_connectres switch (sfsstat status) {
 case SFS_OK:
   sfs_connectok reply;
 case SFS_REDIRECT:
   sfs_pathrevoke revoke;
 default:
   void;
};

struct sfs_encryptarg {
  sfs_hashpay payment;
  sfs_ctext kmsg;
  sfs_pubkey pubkey;
};
typedef sfs_ctext sfs_encryptres;

struct sfs_nfs3_subfs {
  nfspath3 path;
  nfs_fh3 fh;
};
struct sfs_nfs3_fsinfo {
  nfs_fh3 root;
  sfs_nfs3_subfs subfs<>;
};

union sfs_nfs_fsinfo switch (int vers) {
 case ex_NFS_V3:
   sfs_nfs3_fsinfo v3;
};
```

```
const SFSRO_IVSIZE = 16;

#define SFSRO_PROGRAM 344446
#define SFSRO_VERSION 1

struct sfsro1_signed_fsinfo {
  sfs_msgtype type; /* = SFS_ROFSINFO */
  unsigned start;
  unsigned duration;
  opaque iv[SFSRO_IVSIZE];
  sfs_hash rootfh;
  sfs_hash fhdb;
};

struct sfsro1_fsinfo {
  sfsro1_signed_fsinfo info;
  sfs_sig sig;
};

union sfsro_fsinfo switch (int vers) {
 case SFSRO_VERSION:
   sfsro1_fsinfo v1;
};

union sfs_fsinfo switch (int prog) {
 case ex_NFS_PROGRAM:
   sfs_nfs_fsinfo nfs;
 case SFSRO_PROGRAM:
   sfsro_fsinfo sfsro;
 default:
   void;
};


typedef string sfs_idname<32>;

union sfs_opt_idname switch (bool present) {
 case TRUE:
   sfs_idname name;
 case FALSE:
   void;
};

struct sfs_idnums {
  int uid;
  int gid;
};

struct sfs_idnames {
  sfs_opt_idname uidname;
  sfs_opt_idname gidname;
};

enum sfs_loginstat {
```

```
  SFSLOGIN_OK = 0,              /* Login succeeded */
  SFSLOGIN_MORE = 1,            /* More communication with client needed */
  SFSLOGIN_BAD = 2,             /* Invalid login */
  SFSLOGIN_ALLBAD = 3           /* Invalid login don't try again */
};


union sfs_loginres switch (sfs_loginstat status) {
 case SFSLOGIN_OK:
   unsigned authno;
 case SFSLOGIN_MORE:
   opaque resmore<>;
 case SFSLOGIN_BAD:
 case SFSLOGIN_ALLBAD:
   void;
};

struct sfs_loginarg {
  sfs_seqno seqno;
  opaque certificate<>;         /* marshalled sfs_autharg */
};


/*
 * User-authentication structures
 */

enum sfsauth_stat {
  SFSAUTH_OK = 0,
  SFSAUTH_LOGINMORE = 1,        /* More communication with client needed */
  SFSAUTH_FAILED = 2,
  SFSAUTH_LOGINALLBAD = 3,      /* Invalid login don't try again */
  SFSAUTH_NOTSOCK = 4,
  SFSAUTH_BADUSERNAME = 5,
  SFSAUTH_WRONGUID = 6,
  SFSAUTH_DENYROOT = 7,
  SFSAUTH_BADSHELL = 8,
  SFSAUTH_DENYFILE = 9,
  SFSAUTH_BADPASSWORD = 10,
  SFSAUTH_USEREXISTS = 11,
  SFSAUTH_NOCHANGES = 12,
  SFSAUTH_NOSRP = 13,
  SFSAUTH_BADSIGNATURE = 14,
  SFSAUTH_PROTOERR = 15,
  SFSAUTH_NOTTHERE = 16,
  SFSAUTH_BADAUTHID = 17,
  SFSAUTH_KEYEXISTS = 18,
  SFSAUTH_BADKEYNAME = 19
};

enum sfs_authtype {
  SFS_NOAUTH = 0,
  SFS_AUTHREQ = 1
};
```

```
struct sfs_authreq {
  sfs_pubkey usrkey;            /* Key with which signed_req signed */
  sfs_sig signed_req;           /* Recoveraby signed sfs_signed_authreq */
  /* string usrinfo<15>;        /* Logname or "" if any */
};

union sfs_autharg switch (sfs_authtype type) {
 case SFS_NOAUTH:
   void;
 case SFS_AUTHREQ:
   sfs_authreq req;
};

enum sfs_credtype {
  SFS_NOCRED = 0,
  SFS_UNIXCRED = 1
};

struct sfs_unixcred {
  string username<>;
  string homedir<>;
  string shell<>;
  unsigned uid;
  unsigned gid;
  unsigned groups<>;
};

union sfsauth_cred switch (sfs_credtype type) {
 case SFS_NOCRED:
   void;
 case SFS_UNIXCRED:
   sfs_unixcred unixcred;
};

struct sfsauth_loginokres {
  sfsauth_cred cred;
  sfs_hash authid;
  sfs_seqno seqno;
};


union sfsauth_loginres switch (sfs_loginstat status) {
 case SFSLOGIN_OK:
   sfsauth_loginokres resok;
 case SFSLOGIN_MORE:
   opaque resmore<>;
 default:
   void;
};


/*
 * Secure Remote Password (SRP) protocol
```

```
 */

struct sfssrp_parms {
  bigint N;                     /* Prime */
  bigint g;                     /* Generator */
};

union sfsauth_srpparmsres switch (sfsauth_stat status) {
 case SFSAUTH_OK:
   sfssrp_parms parms;
 default:
   void;
};

typedef opaque sfssrp_bytes<>;
struct sfssrp_init_arg {
  string username<>;
  sfssrp_bytes msg;
};

union sfsauth_srpres switch (sfsauth_stat status) {
 case SFSAUTH_OK:
   sfssrp_bytes msg;
 default:
   void;
};

struct sfsauth_fetchresok {
  string privkey<>;
  sfs_hash hostid;
};

union sfsauth_fetchres switch (sfsauth_stat status) {
 case SFSAUTH_OK:
   sfsauth_fetchresok resok;
 default:
   void;
};

struct sfsauth_srpinfo {
  string info<>;
  string privkey<>;
};

struct sfsauth_registermsg {
  sfs_msgtype type;             /* = SFS_AUTHREGISTER */
  string username<>;            /* logname */
  string password<>;            /* password for an add */
  sfs_pubkey pubkey;
  sfsauth_srpinfo *srpinfo;
};

struct sfsauth_registerarg {
  sfsauth_registermsg msg;
```

```
  sfs_sig sig;
};

enum sfsauth_registerres {
  SFSAUTH_REGISTER_OK = 0,
  SFSAUTH_REGISTER_NOTSOCK = 1,
  SFSAUTH_REGISTER_BADUSERNAME = 2,
  SFSAUTH_REGISTER_WRONGUID = 3,
  SFSAUTH_REGISTER_DENYROOT = 4,
  SFSAUTH_REGISTER_BADSHELL = 5,
  SFSAUTH_REGISTER_DENYFILE = 6,
  SFSAUTH_REGISTER_BADPASSWORD = 7,
  SFSAUTH_REGISTER_USEREXISTS = 8,
  SFSAUTH_REGISTER_FAILED = 9,
  SFSAUTH_REGISTER_NOCHANGES = 10,
  SFSAUTH_REGISTER_NOSRP = 11,
  SFSAUTH_REGISTER_BADSIG = 12
};

struct sfsauth_updatemsg {
  sfs_msgtype type;             /* = SFS_AUTHUPDATE */
  sfs_hash authid;              /* SHA-1 (sfs_authinfo);
                                   service is SFS_AUTHSERV */
  sfs_pubkey oldkey;
  sfs_pubkey newkey;
  sfsauth_srpinfo *srpinfo;
  /* maybe username? */
};

struct sfsauth_updatearg {
  sfsauth_updatemsg msg;
  sfs_sig osig;                 /* computed with sfsauth_updatereq.oldkey */
  sfs_sig nsig;                 /* computed with sfsauth_updatereq.newkey */
};

program SFS_PROGRAM {
        version SFS_VERSION {
                void
                SFSPROC_NULL (void) = 0;

                sfs_connectres
                SFSPROC_CONNECT (sfs_connectarg) = 1;

                sfs_encryptres
                SFSPROC_ENCRYPT (sfs_encryptarg) = 2;

                FSINFO
                SFSPROC_GETFSINFO (void) = 3;

                sfs_loginres
                SFSPROC_LOGIN (sfs_loginarg) = 4;

                void
                SFSPROC_LOGOUT (unsigned) = 5;
```

```
                sfs_idnames
                SFSPROC_IDNAMES (sfs_idnums) = 6;

                sfs_idnums
                SFSPROC_IDNUMS (sfs_idnames) = 7;

                sfsauth_cred
                SFSPROC_GETCRED (void) = 8;
        } = 1;
} = 344440;

program SFSCB_PROGRAM {
        version SFSCB_VERSION {
                void
                SFSCBPROC_NULL(void) = 0;
        } = 1;
} = 344441;

program SFSAUTH_PROGRAM {
        version SFSAUTH_VERSION {
                void
                SFSAUTHPROC_NULL (void) = 0;

                sfsauth_loginres
                SFSAUTHPROC_LOGIN (sfs_loginarg) = 1;

                sfsauth_stat
                SFSAUTHPROC_REGISTER (sfsauth_registerarg) = 2;

                sfsauth_stat
                SFSAUTHPROC_UPDATE (sfsauth_updatearg) = 3;

                sfsauth_srpparmsres
                SFSAUTHPROC_SRP_GETPARAMS (void) = 4;

                sfsauth_srpres
                SFSAUTHPROC_SRP_INIT (sfssrp_init_arg) = 5;

                sfsauth_srpres
                SFSAUTHPROC_SRP_MORE (sfssrp_bytes) = 6;

                sfsauth_fetchres
                SFSAUTHPROC_FETCH (void) = 7;
        } = 1;
} = 344442;

#undef SFSRO_VERSION
#undef SFSRO_PROGRAM
```

A.6 Read-only protocol

```
/*
 * This file was written by Frans Kaashoek and Kevin Fu.  Its contents is
 * uncopyrighted and in the public domain.  Of course, standards of
 * academic honesty nonetheless prevent anyone in research from
 * falsely claiming credit for this work.
 */

%#include "bigint.h"
%#include "sfs_prot.h"

const SFSRO_FHSIZE = 20;
const SFSRO_BLKSIZE = 8192;
const SFSRO_NFH = 128;         /* Blocks are approx 2KB each */
const SFSRO_NDIR = 7;
const SFSRO_FHDB_KEYS     = 255;
const SFSRO_FHDB_CHILDREN = 256; /* must be  KEYS+1 */
const SFSRO_FHDB_NFH      = 256; /* FHDB blocks are approx 5KB each */

enum sfsrostat {
  SFSRO_OK = 0,
  SFSRO_ERRNOENT = 1
};

struct sfsro_dataresok {
  uint32 count;
  opaque data<>;
};

union sfsro_datares switch (sfsrostat status) {
 case SFSRO_OK:
   sfsro_dataresok resok;
 default:
   void;
};

enum ftypero {
  SFSROREG      = 1,
  SFSROREG_EXEC = 2,  /* Regular, executable file */
  SFSRODIR      = 3,
  SFSRODIR_OPAQ = 4,
  SFSROLNK      = 5
};


struct sfsro_inode_lnk {
  uint32 nlink;
  nfstime3 mtime;
  nfstime3 ctime;

  nfspath3 dest;
```

```
};

struct sfsro_inode_reg {
  uint32 nlink;
  uint64 size;
  uint64 used;
  nfstime3 mtime;
  nfstime3 ctime;

  sfs_hash direct<SFSRO_NDIR>;
  sfs_hash indirect;
  sfs_hash double_indirect;
  sfs_hash triple_indirect;

};

union sfsro_inode switch (ftypero type) {
 case SFSROLNK:
   sfsro_inode_lnk lnk;
 default:
   sfsro_inode_reg reg;
};


struct sfsro_indirect {
  sfs_hash handles<SFSRO_NFH>;
};

struct sfsro_dirent {
  sfs_hash fh;
  string name<>;
  sfsro_dirent *nextentry;
  /*  uint64 fileid; */
};

struct sfsro_directory {
  nfspath3 path;
/*  uint64 fileid; */
  sfsro_dirent *entries;
  bool eof;
};


struct sfsro_fhdb_indir {
  /*
     Invariant:
                key[i] < key [j] for all i<j

                keys in GETDATA(child[i]) are
                   <= key[i+1] <
                keys in GETDATA(child[i+1])
  */
  sfs_hash key<SFSRO_FHDB_KEYS>;
  sfs_hash child<SFSRO_FHDB_CHILDREN>;
```

```
};

/* Handles to direct blocks */
typedef sfs_hash sfsro_fhdb_dir<SFSRO_FHDB_NFH>;


enum dtype {
   SFSRO_INODE      = 0,
   SFSRO_FILEBLK    = 1, /* File data */
   SFSRO_DIRBLK     = 2, /* Directory data */
   SFSRO_INDIR      = 3, /* Indirect data pointer block */
   SFSRO_FHDB_DIR   = 4, /* Direct data pointer block for FH database */
   SFSRO_FHDB_INDIR = 5  /* Indirect data pointer block for FH database */
};

union sfsro_data switch (dtype type) {
 case SFSRO_INODE:
   sfsro_inode inode;
 case SFSRO_FILEBLK:
   opaque data<>;
 case SFSRO_DIRBLK:
   sfsro_directory dir;
 case SFSRO_INDIR:
   sfsro_indirect indir;
 case SFSRO_FHDB_DIR:
   sfsro_fhdb_dir fhdb_dir;
 case SFSRO_FHDB_INDIR:
   sfsro_fhdb_indir fhdb_indir;
 default:
   void;
};


program SFSRO_PROGRAM {
        version SFSRO_VERSION {
                void
                SFSROPROC_NULL (void) = 0;

                sfsro_datares
                SFSROPROC_GETDATA (sfs_hash) = 1;

        } = 1;
} = 344446;
```

A.7 Agent Protocol

```
/* $Id: sfsagent.x,v 1.29 1999/12/23 02:15:17 dm Exp $ */

/*
 * This file was written by David Mazieres and Michael Kaminsky.  Its
```

```
 * contents is uncopyrighted and in the public domain.  Of course,
 * standards of academic honesty nonetheless prevent anyone in
 * research from falsely claiming credit for this work.
 */

%#include "sfs_prot.h"

typedef string sfs_filename<255>;

struct sfsagent_authinit_arg {
  int ntries;
  string requestor<>;
  sfs_authinfo authinfo;
  sfs_seqno seqno;
};

struct sfsagent_authmore_arg {
  sfs_authinfo authinfo;
  sfs_seqno seqno;
  opaque challenge<>;
};

union sfsagent_auth_res switch (bool authenticate) {
case TRUE:
  opaque certificate<>;
case FALSE:
  void;
};

typedef string sfsagent_path<1024>;
union sfsagent_lookup_res switch (bool makelink) {
 case TRUE:
   sfsagent_path path;
 case FALSE:
   void;
};

enum sfs_revocation_type {
  REVOCATION_NONE = 0,
  REVOCATION_BLOCK = 1,
  REVOCATION_CERT = 2
};
union sfsagent_revoked_res switch (sfs_revocation_type type) {
 case REVOCATION_NONE:
   void;
 case REVOCATION_BLOCK:
   void;
 case REVOCATION_CERT:
   sfs_pathrevoke cert;
};


struct sfsagent_symlink_arg {
  sfs_filename name;
```

```
  sfsagent_path contents;
};

typedef opaque sfsagent_seed[48];
const sfs_badgid = -1;


typedef string sfsagent_comment<1023>;
struct sfs_addkey_arg {
  bigint p;
  bigint q;
  sfs_time expire;
  sfsagent_comment comment;
};

enum sfs_remkey_type {
  SFS_REM_PUBKEY,
  SFS_REM_COMMENT
};
union sfs_remkey_arg switch (sfs_remkey_type type) {
 case SFS_REM_PUBKEY:
   sfs_pubkey pubkey;
 case SFS_REM_COMMENT:
   sfsagent_comment comment;
};

struct sfs_keylistelm {
  bigint key;
  sfs_time expire;
  sfsagent_comment comment;
  sfs_keylistelm *next;
};
typedef sfs_keylistelm *sfs_keylist;

typedef string sfsagent_progarg<>;
typedef sfsagent_progarg sfsagent_cmd<>;

struct sfsagent_certprog {
  string suffix<>;             /* Suffix to be removed from file names */
  string filter<>;             /* Regular expression filter on prefix */
  string exclude<>;            /* Regular expression filter on prefix */
  sfsagent_cmd av;             /* External program to run */
};

typedef sfsagent_certprog sfsagent_certprogs<>;

struct sfsagent_blockfilter {
  string filter<>;             /* Regular expression filter on hostname */
  string exclude<>;            /* Regular expression filter on hostname */
};
struct sfsagent_revokeprog {
  sfsagent_blockfilter *block;  /* Block hostid even without revocation cert */
  sfsagent_cmd av;             /* External program to run */
};
```

```
typedef sfsagent_revokeprog sfsagent_revokeprogs<>;

typedef sfs_hash sfsagent_norevoke_list<>;

struct sfsctl_getfh_arg {
  filename3 filesys;
  u_int64_t fileid;
};

union sfsctl_getfh_res switch (nfsstat3 status) {
 case NFS3_OK:
   nfs_fh3 fh;
 default:
   void;
};

struct sfsctl_getidnames_arg {
  filename3 filesys;
  sfs_idnums nums;
};

union sfsctl_getidnames_res switch (nfsstat3 status) {
 case NFS3_OK:
   sfs_idnames names;
 default:
   void;
};

struct sfsctl_getidnums_arg {
  filename3 filesys;
  sfs_idnames names;
};

union sfsctl_getidnums_res switch (nfsstat3 status) {
 case NFS3_OK:
   sfs_idnums nums;
 default:
   void;
};

union sfsctl_getcred_res switch (nfsstat3 status) {
 case NFS3_OK:
   sfsauth_cred cred;
 default:
   void;
};

struct sfsctl_lookup_arg {
  filename3 filesys;
  diropargs3 arg;
};

program AGENTCTL_PROG {
```

```
        version AGENTCTL_VERS {
                void
                AGENTCTL_NULL (void) = 0;

                bool
                AGENTCTL_ADDKEY (sfs_addkey_arg) = 1;

                bool
                AGENTCTL_REMKEY (sfs_remkey_arg) = 2;

                void
                AGENTCTL_REMALLKEYS (void) = 3;

                sfs_keylist
                AGENTCTL_DUMPKEYS (void) = 4;

                void
                AGENTCTL_CLRCERTPROGS (void) = 5;

                bool
                AGENTCTL_ADDCERTPROG (sfsagent_certprog) = 6;

                sfsagent_certprogs
                AGENTCTL_DUMPCERTPROGS (void) = 7;

                void
                AGENTCTL_CLRREVOKEPROGS (void) = 8;

                bool
                AGENTCTL_ADDREVOKEPROG (sfsagent_revokeprog) = 9;

                sfsagent_revokeprogs
                AGENTCTL_DUMPREVOKEPROGS (void) = 10;

                void
                AGENTCTL_SETNOREVOKE (sfsagent_norevoke_list) = 11;

                sfsagent_norevoke_list
                AGENTCTL_GETNOREVOKE (void) = 12;

                void
                AGENTCTL_SYMLINK (sfsagent_symlink_arg) = 13;

                void
                AGENTCTL_RESET (void) = 14;

                int
                AGENTCTL_FORWARD (sfs_hostname) = 15;

                void
                AGENTCTL_RNDSEED (sfsagent_seed) = 16;
        } = 1;
} = 344428;
```

```
program SETUID_PROG {
        version SETUID_VERS {
                /* Note:  SETUIDPROC_SETUID requires an authunix AUTH. */
                int SETUIDPROC_SETUID (void) = 0;
        } = 1;
} = 344430;

program AGENT_PROG {
        version AGENT_VERS {
                void
                AGENT_NULL (void) = 0;

                int
                AGENT_START (void) = 1;

                int
                AGENT_KILL (void) = 2;

                int
                AGENT_KILLSTART (void) = 3;

                void
                AGENT_SYMLINK (sfsagent_symlink_arg) = 4;

                void
                AGENT_FLUSHNAME (sfs_filename) = 5;

                void
                AGENT_FLUSHNEG (void) = 6;

                void
                AGENT_REVOKE (sfs_pathrevoke) = 7;

                sfsagent_seed
                AGENT_RNDSEED (void) = 8;

                unsigned
                AGENT_AIDALLOC (void) = 9;

                int
                AGENT_GETAGENT (void) = 10;
        } = 1;
} = 344432;

program AGENTCB_PROG {
        version AGENTCB_VERS {
                void
                AGENTCB_NULL (void) = 0;

                sfsagent_auth_res
                AGENTCB_AUTHINIT (sfsagent_authinit_arg) = 1;

                sfsagent_auth_res
                AGENTCB_AUTHMORE (sfsagent_authmore_arg) = 2;
```

```
                sfsagent_lookup_res
                AGENTCB_LOOKUP (sfs_filename) = 3;

                sfsagent_revoked_res
                AGENTCB_REVOKED (filename3) = 4;

                void
                AGENTCB_CLONE (void) = 5;
        } = 1;
} = 344433;

program SFSCTL_PROG {
        version SFSCTL_VERS {
                void
                SFSCTL_NULL (void) = 0;

                void
                SFSCTL_SETPID (int) = 1;

                sfsctl_getfh_res
                SFSCTL_GETFH (sfsctl_getfh_arg) = 2;

                sfsctl_getidnames_res
                SFSCTL_GETIDNAMES (sfsctl_getidnames_arg) = 3;

                sfsctl_getidnums_res
                SFSCTL_GETIDNUMS (sfsctl_getidnums_arg) = 4;

                sfsctl_getcred_res
                SFSCTL_GETCRED (filename3) = 5;

                lookup3res
                SFSCTL_LOOKUP (sfsctl_lookup_arg) = 6;
        } = 1;
} = 344434;
```

A.8 SRP Protocol

```
/* $Id: crypt_prot.x,v 1.3 1999/11/26 06:31:43 dm Exp $ */

/*
 * This file was written by David Mazieres.  Its contents is
 * uncopyrighted and in the public domain.  Of course, standards of
 * academic honesty nonetheless prevent anyone in research from
 * falsely claiming credit for this work.
 */

%#include "bigint.h"
```

```
/*
 * These structures define the raw byte formats of messages exchanged
 * by the SRP protocol, as published in:
 *
 * T. Wu, The Secure Remote Password Protocol, in Proceedings of the
 * 1998 Internet Society Network and Distributed System Security
 * Symposium, San Diego, CA, Mar 1998, pp. 97-111.
 *
 *   sessid is a session identifier known by the user and server to be fresh
 *
 *   N is a prime number such that (N-1)/2 is also prime
 *   g is a generator of Z_N^*
 *
 *   x is a function of the user's password and salt
 *   v is g^x mod N
 *
 *   a is a random element of Z_N^* selected by the user (client)
 *   A = g^a mod N
 *
 *   b and u are random elements of Z_N^* picked by the server
 *   B = v + g^b mod N
 *
 *   S = g^{ab} * g^{xub}
 *   M = SHA-1 (sessid, N, g, user, salt, A, B, S)
 *   H = SHA-1 (sessid, A, M, S)
 *
 * The protocol proceeds as follows:
 *
 *   User -> Server:  username
 *   Server -> User:  salt, N, g
 *   User -> Server:  A
 *   Server -> User:  B, u
 *   User -> Server:  M
 *   Server -> User:  H
 *
 * After this, K can be used to generate secret session keys for use
 * between the user and server.
 */

typedef opaque _srp_hash[20];

/* server to client */
struct srp_msg1 {
  string salt<>;
  bigint N;
  bigint g;
};

/* client to server */
struct srp_msg2 {
  bigint A;
};
```

```
/* server to client */
struct srp_msg3 {
  bigint B;
  bigint u;
};

/* hashed, then client to server */
struct srp_msg4_src {
  _srp_hash sessid;
  bigint N;
  bigint g;
  string user<>;
  string salt<>;
  bigint A;
  bigint B;
  bigint S;
};

/* hashed, then server to client */
struct srp_msg5_src {
  _srp_hash sessid;
  bigint A;
  _srp_hash M;
  bigint S;
};
```

Appendix B

SFS 0.5 User Manual

B.1 Introduction

SFS is a network file system that lets you access your files from anywhere and share them with anyone anywhere. SFS was designed with three goals in mind:

- **Security.** SFS assumes that malicious parties entirely control the network. It ensures that control of the network only lets them delay the file system's operation or conceal the existence of servers until reliable network communication is reestablished.
- **A global namespace.** SFS mounts all remote file systems under the directory '`/sfs`'. The contents of that directory is identical on every client in the world. Clients have no notion of administrative realm and no site-specific configuration options. Servers grant access to users, not to clients. Thus, users can access their files wherever they go, from any machine they trust that runs the SFS client software.
- **Decentralized control.** SFS does not rely on any privileged authority to manage the global namespace. Anyone with a machine on the Internet can set up an SFS file server without needing to obtain any kind of certificates. New servers are instantly accessible from all clients in the world.

SFS achieves these goals by separating key management from file system security. It names file systems by the equivalent of their public keys. Every remote file server is mounted under a directory of the form:

`/sfs/`*Location*:*HostID*

Location is a DNS hostname or an IP address. *HostID* is a collision-resistant cryptographic hash of *Location* and the file server's public key. This naming scheme lets an SFS client authenticate a server given only a file name, freeing the client from any reliance on external key management mechanisms. SFS calls the directories on which it mounts file servers *self-certifying pathnames*.

Self-certifying pathnames let users authenticate servers through a number of different techniques. As a secure, global file system, SFS itself provides a convenient key management infrastructure. Symbolic links let the file namespace double as a key certification namespace. Thus, users can realize many key management schemes using only standard file utilities. Moreover, self-certifying pathnames let people bootstrap one key management mechanism using another, making SFS far more versatile than any file system with built-in key management.

Through a modular implementation, SFS also pushes user authentication out of the file system. Untrusted user processes transparently authenticate users to remote file servers as needed, using protocols opaque to the file system itself.

Finally, SFS separates key revocation from key distribution. Thus, the flexibility SFS provides in key management in no way hinders recovery from compromised keys.

No caffeine was used in the production of the SFS software.

B.2 Installation

This section describes how to build and install the SFS on your system. If you are too impatient to read the details, be aware of the two most important points:

- You must create an '`sfs`' user and an '`sfs`' group on your system. See [–with-sfsuser], page 111, to use a name other than '`sfs`'.

- You must use gcc version 2.95.1 or later to compile SFS.

B.2.1 Requirements

SFS should run with minimal porting on any system that has solid NFS3 support. We have run SFS successfully on OpenBSD 2.6, FreeBSD 3.3, OSF/1 4.0, and Solaris 5.7.

We have also run SFS with some success on Linux. However, you need a kernel with NFS3 support to run SFS on Linux. The SFS on linux web page (`http://www.fs.net/linux/`) has information on installing an SFS-capable Linux kernel.

In order to compile SFS, you will need the following:

1. gcc-2.95.1 or more recent. You can obtain this from `ftp://ftp.gnu.org/pub/gnu/gcc`. Don't waste your time trying to compile SFS with an earlier version of gcc.
2. gmp-2.0.2. You can obtain this from `ftp://ftp.gnu.org/pub/gmp`. Many operating systems already ship with gmp. Note, however, that some Linux distributions do not include the '`gmp.h`' header file. Even if you have libgmp.so, if you don't have /usr/include/gmp.h, you need to install gmp on your system.
3. Header files in '`/usr/include`' that match the kernel you are running. Particularly on Linux where the kernel and user-land utilities are separately maintained, it is easy to patch the kernel without installing the correspondingly patched system header files in '`/usr/include`'. SFS needs to see the patched header files to compile properly.
4. 128 MB of RAM. The C++ compiler really needs a lot of memory.
5. 550 MB of free disk space to build SFS. (Note that on ELF targets, you may be able to get away with considerably less. A build tree on FreeBSD only consumes about 200 MB.)

B.2.2 Building SFS

Once you have setup your system as described in Section B.2.1 [Requirements], page 110, you are ready to build SFS.

1. Create a user, *sfs-user*, and group, *sfs-group*, for SFS on your system. By default, SFS expects the both *sfs-user* and *sfs-group* to be called '`sfs`'. For instance, you might add the following line to '`/etc/passwd`':

   ```
   sfs:*:71:71:Self-certifying file system:/:/bin/true
   ```

   And the following line to '`/etc/group`':

   ```
   sfs:*:71:
   ```

   Do not put any users in *sfs-group*, not even `root`. Any user in *sfs-group* will not be able to make regular use of the SFS file system. Moreover, having an unprivileged users in *sfs-group* causes a security hole.
2. Unpack the SFS sources. For instance, run the commands:

   ```
   % gzip -dc sfs-0.5.tar.gz | tar xvf -
   % cd sfs-0.5
   ```

   If you determined that you need gmp see Section B.2.1 [Requirements], page 110, you should unpack gmp into the top-level of the SFS source tree:

   ```
   % gzip -dc ../gmp-2.0.2.tar.gz | tar xvf -
   ```

3. Set your `CC` and `CXX` environment variables to point to the C and C++ compilers you wish to use to compile SFS. Unless you are using OpenBSD-2.6, your operating system will not come with a recent enough version of gcc see Section B.2.1 [Requirements], page 110.
4. Configure the sources for your system with the command `./configure`. You may additionally specify the following options:

   '`--with-sfsuser=`*sfs-user*'
   : If the user you created for SFS is not called '`sfs`'. Do not use an existing account for *sfs-user*—even a trusted account—as processes running with that user ID will not be able to access SFS. [Note: If you later change your mind about *user-name*, you do not need to recompile SFS, [sfs_config], page 116.]

   '`--with-sfsgroup=`*sfs-group*'
   : If the user you created for SFS does not have the same name as *sfs-user*. [Note: If you later change your mind about *user-group*, you do not need to recompile SFS.]

   '`--with-gmp=`*gmp-path*'
   : To specify where `configure` should look for gmp (for example, *gmp-path* might be '`/usr/local`').

   '`--with-sfsdir=`*sfsdir*'
   : To specify a location for SFS to put its working files. The default is '`/var/sfs`'. [You can change this later, [sfs_config], page 116.]

   '`--with-etcdir=`*etcdir*'
   : To specify where SFS should search for host-specific configuration files. The default is '`/etc/sfs`'.

   `configure` accepts all the traditional GNU configuration options such as '`--prefix`'. It also has several options that are only for developers. **Do not use the '`--enable-repo`' or '`--enable-shlib`' options** (unless you are a gcc maintainer looking for some wicked test cases for your compiler).
5. Build the sources by running '`make`'.
6. Install the binaries by running '`make install`'. If you are short on disk space, you can alternatively install stripped binaries by running '`make install-strip`'.
7. That's it. Fire up the client daemon by running `sfscd`.

B.2.3 Problems building SFS

The most common problem you will encounter is an internal compiler error from gcc. If you are not running gcc-2.95.1 or later, you will very likely experience internal compiler errors when building SFS and need to upgrade the compiler. You must `make clean` after upgrading the compiler. You cannot link object files together if they have been created by different versions of the C++ compiler.

On OSF/1 for the alpha, certain functions using a gcc extension called `__attribute__ ((noreturn))` tend to cause internal compiler errors. If you experience internal compiler errors when compiling SFS for the alpha, try building with the command `make ECXXFLAGS='-D__attribute__\(x\)='` instead of simply `make`.

Sometimes, a particular source file will give particularly stubborn internal compiler errors on some architectures. These can be very hard to work around by just modifying the SFS source code. If you get an internal compiler error you cannot obviously fix, try compiling the particular source file with a different level of debugging. (For example, using a command like `make sfsagent.o CXXDEBUG=-g` in the appropriate subdirectory.)

If your '`/tmp`' file system is too small, you may also end up running out of temporary disk space while compiling SFS. Set your `TMPDIR` environment variable to point to a directory on a file system with more free space (e.g., '`/var/tmp`').

You may need to increase your heap size for the compiler to work. If you use a csh-derived shell, run the command `unlimit datasize`. If you use a Bourne-like shell, run `ulimit -d 'ulimit -H -d'`.

B.3 Getting Started

This chapter gives a brief overview of how to set up an SFS client and server once you have compiled and installed the software.

B.3.1 Quick client setup

SFS clients require no configuration. Simply run the program `sfscd`, and a directory '`/sfs`' should appear on your system. To test your client, access our SFS test server. Type the following commands:

```
% cd /sfs/sfs.fs.net:eu4cvv6wcnzscer98yn4qjpjnn9iv6pi
% 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.

B.3.2 Quick server setup

Setting up an SFS server is a slightly more complicated process. You must perform at least three steps:

1. Create a public/private key pair for your server.
2. Create an '`/etc/sfs/sfsrwsd_config`' configuration file.
3. Configure your machine as an NFS server and export all necessary directories to '`localhost`'.

To create a public/private key pair for your server, run the command:

```
sfskey gen -P /etc/sfs/sfs_host_key
```

Then you must create an '`/etc/sfs/sfsrwsd_config`' file based on which local directories you wish to export and what names those directories should have on clients. This information takes the form of one or more `Export` directives in the configuration file. Each export directive is a line of the form:

```
Export local-directory sfs-name
```

local-directory is the name of a local directory on your system you wish to export. *sfs-name* is the name you wish that directory to have in SFS, relative to the previous `Export` directives. The *sfs-name* of the first `Export` directive must be '`/`'. Subsequent *sfs-names* must correspond to pathnames that already exist in the previously exported directories.

Suppose, for instance, that you wish to export two directories, '`/disk/u1`' and '`/disk/u2`' as '`/usr1`' and '`/usr2`', respectively. You should create a directory to be the root of the exported namespace, say '`/var/sfs/root`', create the necessary *sfs-name* subdirectories, and create a corresponding '`sfsrwsd_config`' file. You might run the following commands to do this:

```
% mkdir /var/sfs/root
% mkdir /var/sfs/root/usr1
% mkdir /var/sfs/root/usr2
```

and create the following '`sfsrwsd_config`' file:

```
Export /var/sfs/root /
Export /disk/u1 /usr1
Export /disk/u2 /usr2
```

Finally, you must export all the *local-directory*s in your '`sfsrwsd_config`' to '`localhost`' via NFS version 3. The details of doing this depend heavily on your operating system. For instance, in OpenBSD you must add the following lines to the file '`/etc/exports`' and run the command '`kill -HUP `cat /var/run/mountd.pid``':

```
/var/sfs/root localhost
/disk/u1 localhost
/disk/u2 localhost
```

On Linux, the syntax for the exports file is:

```
/var/sfs/root localhost(rw)
/disk/u1 localhost(rw)
/disk/u2 localhost(rw)
```

On Solaris, add the following lines to the file '`/etc/dfs/dfstab`' and run '`exportfs -a`':

```
share -F nfs -o -rw=localhost /var/sfs/root
share -F nfs -o -rw=localhost /disk/u1
share -F nfs -o -rw=localhost /disk/u2
```

In general, the procedure for exporting NFS file systems varies greatly between operating systems. Check your operating system's NFS documentation for details. (The manual page for `mountd` is a good place to start.)

Once you have generated a host key, created an '`sfsrwsd_config`' file, and reconfigured your NFS server, you can start the SFS server by running `sfssd`. Note that a lot can go wrong in setting up an SFS server. Thus, we recommend that you first run '`sfssd -d`'. The '`-d`' switch will leave `sfssd` in the foreground and send error messages to your terminal. If there are problems, you can then easily kill `sfssd` from your terminal, fix the problems, and start again. Once things are working, omit the '`-d`' flag; `sfssd` will run in the background and send its output to the system log.

Note: **You will not be able to access an SFS server using the same machine as a client** unless you run `sfscd` with the '`-l`' flag, Section B.6.4 [sfscd], page 128. Attempts to SFS mount a machine on itself will return the error `EDEADLK` (Resource deadlock avoided).

B.3.3 Getting started as an SFS user

To access an SFS server, you must first register a public key with the server, then run the program `sfsagent` on your SFS client to authenticate you.

To register a public key, log into the file server and run the command:

`sfskey register`

This will create a public/private key pair for you and register it 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 address at that server, e.g., '`sfskey register user@other.server.com`'.)

After registering your public key with an SFS server, you must run the `sfsagent` program on an SFS client to access the server. On the client, run the command:

`sfsagent` *user*`@`*server*

server is the name of the server on which you registered, and *user* is your logname on that server. 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 *server* and decrypts it using your passphrase. Finally, it fetches the server's public key, and creates a symbolic link from '`/sfs/`*server*' to '`/sfs/`*server*`:`*HostID*'.

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*`@`*server*

In fact, `sfsagent` runs this exact command for you when you initially start it up.

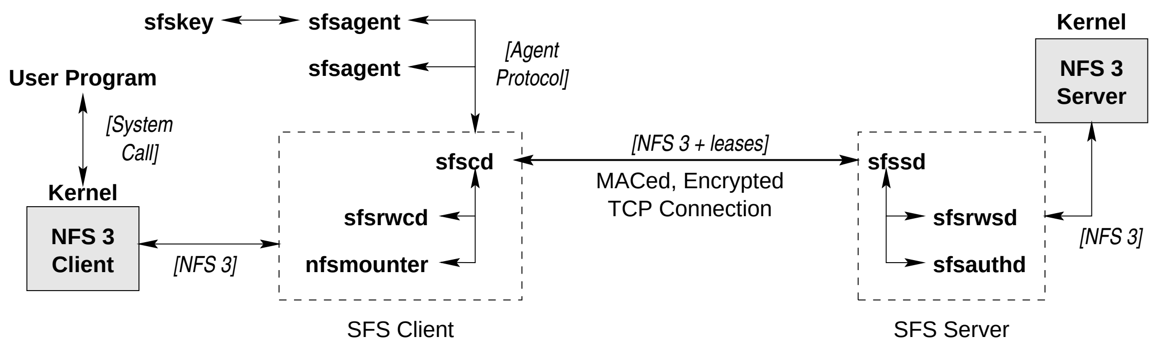
While `sfskey` provides a convenient way of obtaining servers' *HostID*s, it is by no means the only way. Once you have access to one SFS file server, you can store on it symbolic links pointing to other servers' self-certifying pathnames. If you use the same public key on all servers, then, you will only need to type your password once. `sfsagent` will automatically authenticate you to whatever file servers you touch.

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.

B.4 System overview



SFS consists of a number interacting programs on both the client and the server side.

On the client side, SFS implements a file system by pretending to be an NFS server and talking to the local operating system's NFS3 client. The program `sfscd` gets run by root (typically at boot time). `sfscd` spawns two other daemons—`nfsmounter` and `sfsrwcd`.

`nfsmounter` handles the mounting and unmounting of NFS file systems. In the event that `sfscd` dies, `nfsmounter` takes over being the NFS server to prevent file system operations from blocking as it tries to unmount all file systems. **Never send `nfsmounter` a `SIGKILL` signal (i.e., '`kill -9`').** `nfsmounter`'s main purpose is to clean up the mess if any other part of the SFS client software fails. Whatever bad situation SFS has gotten your machine into, killing `nfsmounter` can only make matters worse.

`sfsrwcd` implements the ordinary read-write file system protocol. As other dialects of the SFS protocol become available, they will be implemented as daemons running alongside `sfsrwcd`.

Each user of an SFS client machine must run an instance of the `sfsagent` command. `sfsagent` serves several purposes. It handles user authentication as the user touches new file systems. It can fetch *HostID*s on the fly, a mechanism called *Dynamic server authentication*. Finally, it can perform revocation checks on the *HostID*s of servers the user accesses, to ensure the user does not access *HostID*s corresponding to compromised private keys.

The `sfskey` utility manages both user and server keys. It lets users control and configure their agents. Users can hand new private keys to their agents using `sfskey`, list keys the agent holds, and delete keys. `sfskey` will fetch keys from remote servers using SRP, [SRP], page 123. It lets users change their public keys on remote servers. Finally, `sfskey` can configure the agent for dynamic server authentication and revocation checking.

On the server side, the program `sfssd` spawns two subsidiary daemons, `sfsrwsd` and `sfsauthd`. If virtual hosts or multiple versions of the software are running, `sfssd` may spawn multiple instances of each daemon. `sfssd` listens for TCP connections on port 4. It then hands each connection off to one of the subsidiary daemons, depending on the self-certifying pathname and service requested by the client.

`sfsrwsd` is the server-side counterpart to `sfsrwcd`. It communicates with client side `sfsrwcd` processes using the SFS file system protocol, and accesses the local disk by acting as a client of the local operating system's NFS server. `sfsrwsd` is the one program in sfs that *must be configured* before you run it, [sfsrwsd_config], page 117.

`sfsauthd` handles user authentication. It communicates directly with `sfsrwsd` to authenticate users of the file system. It also accepts connections over the network from `sfskey` to let users download their private keys or change their public keys.

B.5 SFS configuration files

SFS comprises a number of programs, many of which have configuration files. All programs look for configuration files in two directories—first '`/etc/sfs`', then, if they don't find the file there, in '`/usr/local/etc/sfs`'. You can change these locations using the '`--with-etcdir`' and '`--with-confdir`' options to the `configure` command, [configure], page 111.

The SFS software distribution installs reasonable defaults in '`/usr/local/etc/sfs`' for all configuration files except '`sfsrwsd_config`'. On particular hosts where you wish to change the default behavior, you can override the default configuration file by creating a new file of the same name in '/etc/sfs'.

The 'sfs_config' file contains system-wide configuration parameters for most of the programs comprising SFS.

If you are running a server, you will need to create an 'sfsrwsd_config' file to tell SFS what directories to export, and possibly an 'sfsauthd_config' if you wish to share the database of user public keys across several file servers.

The 'sfssd_config' file contains information about which protocols and services to route to which daemons on an SFS server, including support for backwards compatibility across several versions of SFS. You probably don't need to change this file.

'sfs_srp_params' contains some cryptographic parameters for retrieving keys securely over the network with a passphrase (as with the 'sfskey add *usr*@*server*' command).

'sfscd_config' Contains information about extensions to the SFS protocol and which kinds of file servers to route to which daemons. You almost certainly should not touch this file unless you are developing new versions of the SFS software.

Note that configuration command names are case-insensitive in all configuration files (though the arguments are not).

B.5.1 'sfs_config'—system-wide configuration parameters

The 'sfs_config' file lets you set the following system-wide parameters:

'sfsdir *directory*'
: The directory in which SFS stores its working files. The default is '/var/sfs', unless you changed this with the '--with-sfsdir' option to configure.

'sfsuser *sfs-user* [*sfs-group*]'
: As described in Section B.2.2 [Building], page 110, SFS needs its own user and group to run. This configuration directive lets you set the user and group IDs SFS should use. By default, *sfs-user* is 'sfs' and *sfs-group* is the same as *sfs-user*. The 'sfsuser' directive lets you supply either a user and group name, or numeric IDs to change the default. Note: **If you change *sfs-group*, you must make sure the the program '/usr/local/lib/sfs/suidconnect' is setgid to the new *sfs-group*.**

'ResvGids *low-gid high-gid*'
: SFS lets users run multiple instances of the sfsagent program. However, it needs to modify processes' group lists so as to know which file system requests correspond to which agents. The 'ResvGids' directive gives SFS a range of group IDs it can use to tag processes corresponding to a particular agent. (Typically, a range of 16 gids should be plenty.) Note that the range is inclusive—both *low-gid* and *high-gid* are considered reserved gids.

  The setuid root program '/usr/local/lib/sfs/newaid' lets users take on any of these group IDs. Thus, make sure these groups are not used for anything else, or you will create a security hole. There is no default for 'ResvGids'.

'PubKeySize *bits*'
: Sets the default number of bits in a public key. The default value of *bits* is 1280.

'PwdCost *cost*'
: Sets the computational cost of processing a user-chosen password. SFS uses passwords to encrypt users' private keys. Unfortunately, users tend to choose poor passwords. As computers get faster, guessing passwords gets easier. By increasing the *cost* parameter, you can maintain the cost of guessing passwords as hardware improves. *cost* is an exponential parameter. The default value is 7. You probably don't want anything larger than 10. The maximum value is 32—at which point password hashing will not terminate in any tractable amount of time and the `sfskey` command will be unusable.

'LogPriority *facility*.*level*'
: Sets the syslog facility and level at which SFS should log activity. The default is '`daemon.notice`'.

B.5.2 '`sfsrwsd_config`'—File server configuration

'Hostname *name*'
: Set the *Location* part of the server's self-certifying pathname. The default is the current host's fully-qualified hostname.

'Keyfile *path*'
: Tells `sfsrwsd` to look for its private key in file *path*. The default is '`sfs_host_key`'. SFS looks for file names that do not start with '`/`' in '`/etc/sfs`', or whatever directory you specified if you used the '`--with-etcdir`' option to `configure` (see [configure], page 111).

'Export *local-directory sfs-name* `[R|W]`'
: Tells `sfsrwsd` to export *local-directory*, giving it the name *sfs-name* with respect to the server's self-certifying pathname. Appending '`R`' to an export directive gives anonymous users read-only access to the file system (under user ID -2 and group ID -2). Appending '`W`' gives anonymous users both read and write access. See Section B.3.2 [Quick server setup], page 112, for an example of the '`Export`' directive.

  There is almost no reason to use the '`W`' flag. The '`R`' flag lets anyone on the Internet issue NFS calls to your kernel as user -2. SFS filters these calls; it makes sure that they operate on files covered by the export directive, and it blocks any calls that would modify the file system. This approach is safe given a perfect NFS3 implementation. If, however, there are bugs in your NFS code, attackers may exploit them if you have the '`R`' option—probably just crashing your server but possibly doing worse.

'NFSHost *NFS-server*'
: Ordinarily, `sfsrwsd` exports the local file systems of the machine it is running on. However, you can also set up one machine to act as an SFS-proxy for another machine, if you trust the network between the two machines. '`NFSHost`' tells `sfsrwsd` to export file systems from host *NFS-server* instead of from the local machine. All the *local-directory* entries in '`Export`' directives must exist and be NFS-exported from *NFS-server* to the machine actually running `sfsrwsd`.

'LeaseTime *seconds*'

B.5.3 '`sfsauthd_config`'—User-authentication daemon configuration

'`Hostname` *name*'
: Set the *Location* part of the server's self-certifying pathname. The default is the current host's fully-qualified hostname.

'`Keyfile` *path*'
: Tells `sfsrwsd` to look for its private key in file *path*. The default is '`sfs_host_key`'. SFS looks for file names that do not start with '`/`' in '`/etc/sfs`', or whatever directory you specified if you used the '`--with-etcdir`' option to `configure` (see [configure], page 111).

'`Userfile [-ro|-reg] [-pub=`*pubpath*`] [-mapall=`*user*`]` *path*'
: This specifies a file in which `sfsauthd` should look for user public keys when authenticating users. You can specify multiple '`Userfile`' directives to use multiple files. This can be useful in an environment where most user accounts are centrally maintained, but a particular server has a few locally-maintained guest (or root) accounts.

  Userfile has the following options:

  '`-ro`'
  : Specifies a read-only user database—typically a file on another SFS server. `sfsauthd` will not allow users in a read-only database to update their public keys. It also assumes that read-only databases reside on other machines. Thus, it maintains local copies of read-only databases in '`/var/sfs/authdb`'. This process ensures that temporarily unavailable file servers never disrupt `sfsauthd`'s operation.

  '`-reg`'
  : Allows users who do not exist in the database to register initial public keys by typing their UNIX passwords. See [sfskey register], page 126, for details on this. At most one '`Userfile`' can have the '`-reg`' option. '`-reg`' and '`-ro`' are mutually exclusive.

  '`-pub=`*pubpath*'
  : `sfsauthd` supports the secure remote password protocol, or SRP. SRP lets users connect securely to `sfsauthd` with their passwords, without needing to remember the server's public key. To prove its identity through SRP, the server must store secret data derived from a user's password. The file *path* specified in '`Userfile`' contains these secrets for users opting to use SRP. The '`-pub`' option tells `sfsauthd` to maintain in *pubpath* a separate copy of the database without secret information. *pubpath* might reside on an anonymously readable SFS file system—other machines can then import the file as a read-only database using the '`-ro`' option.

  '`-mapall=`*user*'
  : Map every entry in the user database to the the local user *user*, regardless of the actual credentials specified by the file.

  If no '`Userfile`' directive is specified, `sfsauthd` uses the following default (again, unqualified names are assumed to be in '`/etc/sfs`'):

```
Userfile -reg -pub=sfs_users.pub sfs_users
```

'SRPfile *path*'
: Where to find default parameters for the SRP protocol. The default is 'sfs_srp_params'.

'Denyfile *path*'
: Specifies a file listing users who should not be able to register public keys with 'sfskey register'. The default is 'sfs_deny'.

B.5.4 'sfs_users'—User-authentication database

The 'sfs_users' file, maintained and used by the sfsauthd program, maps public keys to local users. It is roughly analogous to the Unix '/etc/passwd' file. Each line of 'sfs_users' has the following format:

user:*public-key*:*credentials*:*SRP-info*:*private-key*

user — *user* is the unique name of a public key in the database. Ordinarily it the same as a user-name in the local password file. However, in certain cases it may be useful to map multiple public keys to the same local account (for instance if several people have an account with root privileges). In such cases, each key should be given a unique name (e.g., 'dm/root', 'kaminsky/root', etc.).

public-key — Public key is simply the user's public key. A user must posses the corresponding private key to authenticate himself to servers.

credentials
: *credentials* are the credentials associated with a particular SFS public key. It is simply a local username to be looked up in the Unix password and group databases. Ordinarily, *credentials* should be the same as *user* unless multiple keys need to be mapped to the same *credentials*.

SRP-info — *SRP-info* is the server-side information for the SRP protocol, [SRP], page 123. Unlike the previous fields, this information must be kept secret. If the information is disclosed, an attacker may be able to impersonate the server by causing the sfskey add command to fetch the wrong *HostID*. Note also that *SRP-info* is specific to a particular hostname. If you change the *Location* of a file server, users will need to register new *SRP-info*.

private-key
: *private-key* is actually opaque to sfsauthd. It is private, per-user data that sfsauthd will return to users who successfully complete the SRP protocol. Currently, sfskey users this field to store an encrypted copy of a user's private key, allowing the user to retrieve the private key over the network.

B.5.5 'sfssd_config'—Meta-server configuration

'sfssd_config' configures sfssd, the server that accepts connections for sfsrwsd and sfsauthd. 'sfssd_config' can be used to run multiple "virtual servers", or to run several versions of the server software for compatibility with old clients.

Directives are:

'RevocationDir *path*'
: Specifies the directory in which sfssd should search for revocation/redirection certificates when clients connect to unknown (potentially revoked) self-certifying pathnames. The default value is '/var/sfs/srvrevoke'. Use the command 'sfskey revokegen' to generate revocation certificates.

'HashCost *bits*'
: Specifies that clients must pay for connections by burning CPU time. This can help reduce the effectiveness of denial-of-service attacks. The default value is 0. The maximum value is 22.

'Server {* | *Location*[:*HostID*]}'
: Specifies a section of the file that applies connection requests for the self-certifying pathname *Location*':'*HostID*. If ':'*HostID* is omitted, then the following lines apply to any connection that does not match an explicit *HostID* in another 'Server'. The argument '*' applies to all clients who do not have a better match for either *Location* or *HostID*.

'Release {* | *sfs-version*}'
: Begins a section of the file that applies to clients running SFS release *sfs-version* or older. '*' signifies arbitrarily large SFS release numbers. The 'Release' directive does not do anything on its own, but applies to all subsequent 'Service' directives until the next 'Release' or 'Server' directive.

'Extensions *ext1* [*ext2* ...]'
: Specifies that subsequent 'Service' directives apply only to clients that supply all of the listed extension strings (*ext1*, ...). 'Extensions' until the next 'Extensions', 'Release' or 'Server' directive

'Service *srvno daemon* [*arg* ...]'
: Specifies the daemon that should handle clients seeking service number *srvno*. SFS defines the following values of *srvno*:

  1. File server
  2. Authentication server
  3. Remote execution (not yet released)
  4. SFS/HTTP (not yet released)

The default contents of 'sfssd_config' is:

```
Server *
  Release *
      Service 1 sfsrwsd
      Service 2 sfsauthd
```

To run a different server for sfs-0.3 and older clients, you could add the lines:

```
Release 0.3
  Service 1 /usr/local/lib/sfs-0.3/sfsrwsd
```

B.5.6 'sfs_srp_params'—Default parameters for SRP protocol

Specifies a "strong prime" and a generator for use in the SRP protocol. SFS ships with a particular set of parameters because generating new ones can take a considerable amount of CPU time. You can replace these parameters with randomly generated ones using the '`sfskey srpgen -b` *bits*' command.

Note that SRP parameters can afford to be slightly shorter than Rabin public keys, both because SRP is based on discrete logs rather than factoring, and because SRP is used for authentication, not secrecy. 1,024 is a good value for *bits* even if '`PubKeySize`' is slightly larger in '`sfs_config`'.

B.5.7 '`sfscd_config`'—Meta-client configuration

The '`sfscd_config`' is really part of the SFS protocol specification. If you change it, you will no longer be executing the SFS protocol. Nonetheless, you need to do this to innovate, and SFS was designed to make implementing new kinds of file systems easy.

'`sfscd_config`' takes the following directives:

'`Extension` *string*'
: Specifies that `sfscd` should send *string* to all servers to advertise that it runs an extension of the protocol. Most servers will ignore *string*, but those that support the extension can pass off the connection to a new "extended" server daemon. You can specify multiple '`Extension`' directives.

'`Release {* |` *sfs-version*`}`'
: Begins a section of the file that applies to servers running SFS release *sfs-version* or older. '`*`' signifies arbitrarily large SFS release numbers. The '`Release`' directive does not do anything on its own, but applies to all subsequent '`Program`' directives until the next '`Release`' directive.

'`Libdir` *path*'
: Specifies where SFS should look for daemon programs when their pathnames do not begin with '`/`'. The default is '`/usr/local/lib/sfs-0.5`'. The '`Libdir`' directive does not do anything on its own, but applies to all subsequent '`Program`' directives until the next '`Libdir`' or '`Release`' directive.

'`Program` *prog*`.`*vers daemon* `[`*arg* `...]`'
: Specifies that connections to servers running Sun RPC program number *prog* and version *vers* should be handed off to the the local daemon *daemon*. SFS currently defines two RPC program numbers. Ordinary read-write servers use program number 344444, version 3 (a protocol very similar to NFS3), while read-only servers use program 344446, version 1. The read-only code has not been released yet. The '`Program`' directive must be preceded by a '`Release`' directive.

The default '`sfscd_config`' file is:

```
Release *
  Program 344444.3 sfsrwcd
```

To run a different set of daemons when talking to sfs-0.3 or older servers, you could add the following lines:

```
Release 0.3
  Libdir /usr/local/lib/sfs-0.3
  Program 344444.3 sfsrwcd
```

B.6 Command reference guide

B.6.1 `sfsagent` reference guide

`sfsagent` is the program users run to authenticate themselves to remote file servers, to create symbolic links in '`/sfs`' on the fly, and to look for revocation certificates. Many of the features in `sfsagent` are controlled by the `sfskey` program and described in the `sfskey` documentation.

Ordinarily, a user runs `sfsagent` at the start of a session. `sfsagent` runs `sfskey add` to obtain a private key. As the user touches each SFS file server for the first time, the agent authenticates the user to the file server transparently using the private key it has. At the end of the session, the user should run `sfskey kill` to kill the agent.

The usage is as follows:

`sfsagent [-dnkF] -S` *sock* `[-c [`*prog* `[`*arg* `...]] |` *keyname*`]`

'`-d`' Stay in the foreground rather than forking and going into the background

'`-n`' Do not attempt to communicate with the SFS file system. This can be useful for debugging, or for running an agent on a machine that is not running an SFS client. If you specify '`-n`', you must also use the '`-S`' option, otherwise your agent will be useless as there will be no way to communicate with it.

'`-k`' Atomically kill and replace any existing agent. Otherwise, if your agent is already running, `sfsagent` will refuse to run again.

'`-F`' Allow forwarding. This will allow programs other than the file system to ask the agent to authenticate the user.

'`-S` *sock*' Listen for connections from programs like `sfskey` on the Unix domain socket *sock*. Ordinarily `sfskey` connects to the agent through the client file system software, but it can use a named Unix domain socket as well.

'`-c [`*prog* `[`*arg* `...]]`'

By default, `sfsagent` on startup runs the command '`sfskey add`' giving it whatever '`-t`' option and *keyname* you specified. This allows you to fetch your first key as you start or restart the agent. If you wish to run a different program, you can specify it using '`-c`'. You might, for instance, wish to run a shell-script that executes a '`sfskey add`' followed by several '`sfskey certprog`' commands.

`sfsagent` runs the program with the environment variable `SFS_AGENTSOCK` set to '`-0`' and a Unix domain socket on standard input. Thus, when atomically killing and restarting the agent using '`-k`', the commands run by `sfsagent` talk to the new agent and not the old.

If you don't wish to run any program at all when starting `sfsagent`, simply supply the '`-c`' option with no *prog*. This will start an new agent that has no private keys.

B.6.2 `sfskey` reference guide

The `sfskey` command performs a variety of key management tasks, from generating and updating keys to controlling users' SFS agents. The general usage for `sfskey` is:

`sfskey [-S` *sock*`] [-p` *pwfd*`]` *command* `[`*arg* `...]`

'`-S`' specifies a UNIX domain socket `sfskey` can use to communicate with your `sfsagent` socket. If *sock* begins with '`-`', the remainder is interpreted as a file descriptor number. The default is to use the environment variable `SFS_AGENTSOCK` if that exists. If not, `sfskey` asks the file system for a connection to the agent.

The '`-p`' option specifies a file descriptor from which `sfskey` should read a passphrase, if it needs one, instead of attempting to read it from the user's terminal. This option may be convenient for scripts that invoke `sfskey`. For operations that need multiple passphrases, you must specify the '`-p`' option multiple times, once for each passphrase.

'`sfskey add [-t [hrs:]min] [`*keyfile*`]`'
'`sfskey add [-t [hrs:]min] [`*user*`]@`*hostname*'

The `add` command loads and decrypts a private key, and gives the key to your agent. Your agent will use it to try to authenticate you to any file systems you reference. The '`-t`' option specifies a timeout after which the agent should forget the private key.

In the first form of the command, the key is loaded from file *keyfile*. The default for *keyfile*, if omitted, is '`$HOME/.sfs/identity`'.

The second form of the command fetches a private key over the network using the SRP (`http://srp.stanford.edu/srp/`) protocol. SRP lets users establish a secure connection to a server without remembering its public key. Instead, to prove their identities to each other, the user remembers a secret password and the server stores a one-way function of the password (also a secret). SRP addresses the fact that passwords are often poorly chosen; it ensures that an attacker impersonating one of the two parties cannot learn enough information to mount an off-line password guessing attack—in other words, the attacker must interact with the server or user on every attempt to guess the password.

The `sfskey update` and `register` commands let users store their private keys on servers, and retrieve them using the `add` command. The private key is stored in encrypted form, using the same password as the SRP protocol (a safe design as the server never sees any password-equivalent data).

Because the second form of `sfskey add` establishes a secure connection to a server, it also downloads the servers HostID securely and creates a symbolic link from '`/sfs/`'*hostname* to the server's self-certifying pathname.

When invoking `sfskey add` with the SRP syntax, `sfskey` will ask for the user's password with a prompt of the following form:

`Passphrase for` *user*`@`*servername*`/`*nbits*`:`

user is simply the username of the key being fetched from the server. *servername* is the name of the server on which the user registerd his SRP information. It may not be the same as the *hostname* argument to `sfskey` if the user has supplied a hostname alias (or CNAME) to `sfskey add`. Finally, *nbits* is the size of the prime number used in the SRP protocol. Higher values are more secure; 1,024 bits should be adequate. However, users should expect always to see the same value for *nbits* (otherwise, someone may be trying to impersonate the server).

'sfskey certclear'
: Clears the list of certification programs the agent runs. See [certprog], page 124, for more details on certification programs.

'sfskey certlist [-q]'
: Prints the list of certification programs the agent runs. See [certprog], page 124, for more details on certification programs.

'sfskey certprog [-s *suffix*] [-f *filter*] [-e *exclude*] *prog* [*arg* ...]'
: The `certprog` command registers a command to be run to lookup *HostIDs* on the fly in the '/sfs' directory. This mechanism can be used for *dynamic server authentication*—running code to lookup *HostIDs* on-demand. When you reference the file '/sfs/*name*.*suffix*', your agent will run the command:

    prog arg ... *name*

  If the program succeeds and prints *dest* to its standard output, the agent will then create a symbolic link:

    /sfs/*name*.*suffix* -> *dest*

  If the '-s' flag is omitted, then neither '.' nor *suffix* gets appended to *name*. In other words, the link is '/sfs/*name* -> *dest*'. *filter* is a perl-style regular expression. If it is specified, then *name* must contain it for the agent to run *prog*. *exclude* is another regular expression, which, if specified, prevents the agent from running *prog* on *names* that contain it (regardless of *filter*).

  The program `dirsearch` can be used with `certprog` to configure *certification paths*—lists of directories in which to look for symbolic links to *HostIDs*. The usage is:

    dirsearch [-clpq] *dir1* [*dir2* ...] *name*

  `dirsearch` searches through a list of directories *dir1*, *dir2*, ... until it finds one containing a file called *name*, then prints the pathname '*dir*/*name*'. If it does not find a file, `dirsearch` exits with a non-zero exit code. The following options affect `dirsearch`'s behavior:

  '-c' Print the contents of the file to standard output, instead of its pathname.

  '-l' Require that '*dir*/*name*' be a symbolic link, and print the path of the link's destination, rather than the path of the link itself.

  '-p' Print the path '*dir*/*name*'. This is the default behavior anyway, so the option '-p' has no effect.

  '-q' Do not print anything. Exit abnormally if *name* is not found in any of the directories.

  As an example, to lookup self-certifying pathnames in the directories '$HOME/.sfs/known_hosts' and '/mit', but only accepting links in '/mit' with names ending '.mit.edu', you might execute the following commands:

```
% sfskey certprog dirsearch $HOME/.sfs/known_hosts
% sfskey certprog -f '\.mit\.edu$' /mnt/links
```

'`sfskey delete` *keyname*'

Deletes private key *keyname* from the agent (reversing the effect of an `add` command).

'`sfskey deleteall`'

Deletes all private keys from the agent.

'`sfskey edit -P [-o` *outfile*`] [-c` *cost*`] [-n` *name*`] [`*keyname*`]`'

Changes the passphrase, passphrase "cost", or name of a public key. Can also download a key from a remote server via SRP and store it in a file.

keyname can be a file name, or it can be of the form '`[`*user*`]@`*server*', in which case `sfskey` will fetch the key remotely and *outfile* must be specified. If *keyname* is unspecified, the default is '`$HOME/.sfs/identity`'.

The options are:

'`-P`' Removes any password from the key, so that the password is stored on disk in unencrypted form.

'`-o` *outfile*'

Specifies the file two which the edited key should be written.

'`-c` *cost*' Override the default computational cost of processing a password, or '`PwdCost`', [pwdcost], page 117.

'`-n` *name*' Specifies the name of the key that shows up in `sfskey list`.

'`sfskey gen [-KP] [-b` *nbits*`] [-c` *cost*`] [-n` *name*`] [`*keyfile*`]`'

Generates a new public/private key pair and stores it in *keyfile*. It omitted *keyfile* defaults to '`$HOME/.sfs/identity`'.

'`-K`' By default, `sfskey gen` asks the user to type random text with which to seed the random number generator. The '`-K`' option suppresses that behavior.

'`-P`' Specifies that `sfskey gen` should not ask for a passphrase and the new key should be written to disk in unencrypted form.

'`-b` *nbits*' Specifies that the public key should be *nbits* long.

'`-c` *cost*' Override the default computational cost of processing a password, or '`PwdCost`', [pwdcost], page 117.

'`-n` *name*' Specifies the name of the key that shows up in `sfskey list`. Otherwise, the user will be prompted for a name.

'`sfskey help`'

Lists all of the various `sfskey` commands and their usage.

'`sfskey hostid` *hostname*'

'`sfskey hostid -`'

Retrieves a self-certifying pathname insecurely over the network and prints '*Location*`:`*HostID*' to standard output. If *hostname* is simply '`-`', returns the name of the current machine, which is not insecure.

'`sfskey kill`'

Kill the agent.

'sfskey list [-ql]'

List the public keys whose private halves the the agent holds.

'-q' Suppresses the banner line explaining the output.

'-l' Lists the actual value of public keys, in addition the the names of the keys.

'sfskey norevokeset *HostID* ...'

'sfskey norevokelist'

'sfskey register [-KS] [-b *nbits*] [-c *cost*] [-u *user*] [*key*]'

The `sfskey register` command lets users who are logged into an SFS file server register their public keys with the file server for the first time. Subsequent changes to their public keys can be authenticated with the old key, and must be performed using `sfskey update`. The superuser can also use `sfskey register` when creating accounts.

key is the private key to use. If *key* does not exist and is a pathname, `sfskey` will create it. The default *key* is '$HOME/.sfs/identity', unless '-u' is used, in which case the default is to generate a new key but not store it anywhere. If a user wishes to reuse a public key already registered with another server, the user can specify '*user*@*server*' for *key*.

'-K'

'-b *nbits*'

'-c *cost*' These options are the same as for `sfskey gen`. '-K' and '-b' have no effect if the key already exists.

'-S' Do not register any SRP information with the server—this will prevent the user from using SRP to connect to the server, but will also prevent the server from gaining any information that could be used by an attacker to mount an off-line guessing attack on the user's password.

'-u *user*' When `sfskey register` is run as root, specifies a particular user to register. This can be useful when creating accounts for people.

'sfsauthd_config' must have a 'Userfile' with the '-reg' option to enable use of the `sfskey register`, [sfsauthd_config], page 118.

'sfskey reset'

Clear the contents of the '/sfs' directory, including all symbolic links created by `sfskey certprog` and `sfskey add`, and log the user out of all file systems.

Note that this is not the same as deleting private keys held by the agent (use `deleteall` for that). In particular, the effect of logging the user out of all file systems will likely not be visible—the user will automatically be logged in again on-demand.

'sfskey revokegen [-r *newkeyfile* [-n *newhost*]] [-o *oldhost*] *oldkeyfile*'

'sfskey revokelist'

'sfskey revokeclear'

'sfskey revokeprog [-b [-f *filter*] [-e *exclude*]] *prog* [*arg* ...]'

'sfskey srpgen [-b *nbits*] file'
: Generate a new 'sfs_srp_params' file, [sfs_srp_params], page 120.

'sfskey update [-S | -s *srp_params*] [-a {*server* | -}] *oldkey* [*newkey*]'
: Change a user's public key and SRP information on an SFS file server. The default value for *newkey* is '$HOME/.sfs/identity'.

  To change public keys, typically a user should generate a new public key and store it in '$HOME/.sfs/identity'. Then he can run 'sfskey update [*user*]@*host*' for each server on which he needs to change his public key.

  Several options control sfskey update's behavior:

  '-S' Do not send SRP information to the server—this will prevent the user from using SRP to connect to the server, but will also prevent the server from gaining any information that could be used by an attacker to mount an off-line guessing attack on the user's password.

  '-s' *srp_params* is the path of a file generated by sfskey srpgen, and specifies the parameters to use in generating SRP information for the server. The default is to get SRP parameters from the server, or look in '/usr/local/etc/sfs/sfs_srp_params'.

  '-a *server*'
  '-a -' Specify the server on which to change the users key. The server must be specified as '*Location*:*HostID*'. A *server* of '-' means to use the local host. You can specify the '-a' option multiple times, in which case sfskey will attempt to change *oldkey* to *newkey* on multiple servers in parallel.

  If *oldkey* is the name of a remote key—i.e. of the form '[*user*]@*host*'—then the default value of *server* is to use whatever server successfully completes the SRP authentication protocol while fetching *oldkey*. Otherwise, if *oldkey* is a file, the '-a' option is mandatory.

B.6.3 ssu command

The ssu command allows an unprivileged user to become root on the local machine without changing his SFS credentials. ssu invokes the command su to become root. Thus, the access and password checks needed to become root are identical to those of the local operating system's su command. ssu also runs '/usr/local/lib/sfs-0.5/newaid' to alter the group list so that SFS can recognize the root shell as belonging to the original user.

The usage is as follows:

```
ssu [-f | -m | -l | -c command]
```

'-f'
'-m' These options are passed through to the su command.

'-l' This option causes the newly spawned root shell to behave like a login shell.

'-c *command*'
: Tells ssu to tell su to run *command* rather than running a shell.

Note, `ssu` does not work on some versions of Linux because of a bug in Linux. To see if this bug is present, run the command '`su root -c ps`'. If this command stops with a signal, your `su` command is broken and you cannot use `ssu`.

B.6.4 `sfscd` command

`sfscd [-d] [-l] [-L] [-f` *config-file*`]`

`sfscd` is the program to create and serve the '`/sfs`' directory on a client machine. Ordinarily, you should not need to configure `sfscd` or give it any command-line options.

'`-d`' Stay in the foreground and print messages to standard error rather than redirecting them to the system log.

'`-l`' Ordinarily, `sfscd` will disallow access to a server running on the same host. If the *Location* in a self-certifying pathname resolves to an IP address of the local machine, any accesses to that pathname will fail with the error `EDEADLK` ("Resource deadlock avoided").

The reason for this behavior is that SFS is implemented using NFS. Many operating systems can deadlock when there is a cycle in the mount graph—in other words when two machines NFS mount each other, or, more importantly when a machine NFS mounts itself. To allow a machine to mount itself, you can run `sfscd` with the '`-l`' flag. This may in fact work fine and not cause deadlock on non-BSD systems.

'`-L`' On Linux, the '`-L`' option disables a number of kludges that work around bugs in the kernel. '`-L`' is useful for people interested in improving Linux's NFS support.

'`-f` *config-file*'

Specify an alternate `sfscd` configuration file, [sfscd_config], page 121. The default, if '`-f`' is unspecified, is first to look for '`/etc/sfs/sfscd_config`', then '`/usr/local/etc/sfs/sfscd_config`'.

B.6.5 `sfssd` command

`sfssd [-d] [-f` *config-file*`]`

`sfssd` is the main server daemon run on SFS servers. `sfssd` itself does not serve any file systems. Rather, it acts as a meta-server, accepting connections on TCP port 4 and passing them off to the appropriate daemon. Ordinarily, `sfssd` passes all file system connections to `sfsrwsd`, and all user-key management connections to `sfsauthd`. However, the '`sfssd_config`' file (see [sfssd_config], page 119) allows a great deal of customization, including support for "virtual servers," multiple versions of the SFS software coexisting, and new SFS-related services other than the file system and user authentication.

'`-d`' Stay in the foreground and print messages to standard error rather than redirecting them to the system log.

'`-f` *config-file*'

Specify an alternate `sfssd` configuration file, [sfssd_config], page 119. The default, if '`-f`' is unspecified, is first to look for '`/etc/sfs/sfssd_config`', then '`/usr/local/etc/sfs/sfssd_config`'.

B.6.6 `sfsrwsd` command

`/usr/local/lib/sfs-0.5/sfsrwsd [-f` *config-file*`]`

`sfsrwsd` is the program implementing the SFS read-write server. Ordinarily, you should never run `sfsrwsd` directly, but rather have `sfssd` do so. Nonetheless, you must create a configuration file for `sfsrwsd` before running an SFS server. See [sfsrwsd_config], page 117, for what to put in your '`sfsrwsd_config`' file.

'`-f` *config-file*'
: Specify an alternate `sfsrwsd` configuration file, [sfsrwsd_config], page 117. The default, if '`-f`' is unspecified, is '`/etc/sfs/sfsrwsd_config`'.

B.7 Security considerations

SFS shares files between machines using cryptographically protected communication. 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, there will very likely be security holes attackers can exploit because of SFS, that they could not have exploited otherwise. This chapter enumerates some of the security consequences of running SFS. The first section describes vulnerabilities that may result from the very existence of a global file system. The next section lists bugs potentially present in your operating system that may be much easier for attackers to exploit if you run SFS. Finally the last section attempts to point out weak points of the SFS implementation that may lead to vulnerabilities in the SFS software itself.

B.7.1 Vulnerabilities created by SFS

Facilitating exploits

Many security holes can be exploited much more easily if the attacker can create an arbitrary file on your system. As a simple example, if a bug allows attackers to run any program on your machine, SFS allows them to supply the program somewhere under '`/sfs`'. Moreover, the file can have any numeric user and group (though of course, SFS disables setuid and devices).

'`.`' in `path`

Another potential problem users putting the current working directory '`.`' in their *PATH* environment variables. If you are browsing a file system whose owner you do not trust, that owner can run arbitrary code as you by creating programs named things like `ls` in the directories you are browsing. Putting '`.`' in the *PATH* has always been a bad idea for security, but a global file system like SFS makes it much worse.

symbolic links from untrusted servers

Users need to be careful about using untrusted file systems as if they were trusted file systems. Any file system can name files in any other file system by symbolic links. Thus, when randomly overwriting files in a file system you do not trust, you can be tricked, by symbolic links, into overwriting files on the local disk or another SFS file system.

As an example of a seemingly appealing use of SFS that can cause problems, consider doing a `cvs` checkout from an untrusted CVS repository, so as to peruse someone else's source code. If you run `cvs` on a repository you do not trust, the person hosting the repository could replace the '`CVSROOT/history`' with a symbolic link to a file on some other file system, and cause you to append garbage to that file.

This `cvs` example may or may not be a problem. For instance, if you are about to compile and run the software anyway, you are placing quite a bit of trust in the person running the CVS repository anyway. The important thing to keep in mind is that for most uses of a file system, you are placing some amount of trust in in the file server.

See [resvgids], page 116, to see how users can run multiple agents with the `newaid` command. One way to cut down on trust is to access untrusted file servers under a different agent with different private keys. Nonetheless, this still allows the remote file servers to serve symbolic links to the local file system in unexpected places.

Leaking information

Any user on the Internet can get the attributes of a *local-directory* listed in an '`Export`' directive (see [export], page 117). This is so users can run commands like '`ls -ld`' on a self-certifying pathname in '`/sfs`', even if they cannot change directory to that pathname or list files under it. If you wish to keep attribute information secret on a *local-directory*, you will need to export a higher directory. We may later reevaluate this design decision, though allowing such anonymous users to get attributes currently simplifies the client implementation.

B.7.2 Vulnerabilities exploitable because of SFS

NFS server security

The SFS read-write server software requires each SFS server to run an NFS server. Running an NFS server at all can constitute a security hole. In order to understand the full implications of running an SFS server, you must also understand NFS security.

NFS security relies on the secrecy of file handles. Each file on an exported file system has associated with it an NFS file handle (typically 24 to 32 bytes long). When mounting an NFS file system, the `mount` command on the client machine connects to a program called `mountd` on the server and asks for the file handle of the root of the exported file system. `mountd` enforces access control by refusing to return this file handle to clients not authorized to mount the file system.

Once a client has the file handle of a directory on the server, it sends NFS requests directly to the NFS server's kernel. The kernel performs no access control on the request (other than checking that the user the client claims to speak for has permission to perform the requested operation). The expectation is that all clients are trusted to speak for all users, and no machine can obtain a valid NFS file handle without being an authorized NFS client.

To prevent attackers from learning NFS file handles when using SFS, SFS encrypts all NFS file handles with a 20-byte key using the Blowfish encryption algorithm. Unfortunately, not all operating systems choose particularly good NFS file handles in the first place. Thus, attackers may be able to guess your file handles anyway. In general, NFS file handles contain the following 32-bit words:

- A file system ID (containing the device number)
- The inode number (i-number) of the file
- A generation number that changes when the i-number is recycled

In addition NFS file handles can contain the following words:

- A second file system ID word (for a 64-bit fsid)
- The length of the file handle data
- The i-number of the exported directory
- The generation number of the exported directory
- Another copy of the file system ID (for the exported directory?)
- One or more unused 0 words

Many of these words can be guessed outright by attackers without their needing to interact with any piece of software on the NFS server. For instance, the file system ID is often just the device number on which the physical file system resides. The i-number of the root directory in a file system is always 2. The i-number and generation number of the root directory can also be used as the i-number and generation number of the "exported directory".

On some operating systems, then, the only hard thing for an attacker to guess is the 32-bit generation number of some directory on the system. Worse yet, the generation numbers are sometimes not chosen with a good random number generator.

To minimize the risks of running an NFS server, you might consider taking the following precautions:

- Many operating systems ship with a program called `fsirand` that re-randomizes all generation numbers in a file system. Running `fsirand` may result in much better generation numbers than, say, a factory install of an operating system.
- In general, you should try to block all external NFS traffic from reaching your machine. If you have a firewall, consider filtering ports 111 and 2049 for both TCP and UDP. If your server's operating system comes with some sort of IP filtering, you might filter any traffic to port 2049 that does not come from the loopback interface (though on some OSes, this could prevent you from acting as an NFS client if you are still using NFS on your local network—try it to see).
- Most operating systems allow you to export NFS file systems "read-mostly"—i.e. read-write to a small number of servers and read-only to everyone else. The read-only requirement typically is enforced by the kernel. Thus, if you can export file systems read-write to '`localhost`' for SFS, but read-only to any client on which an attacker may have learned an NFS file handle, you may be able to protect the integrity of your file system under attack. (Note, however, that unless you filter forged packets at your firewall, the attacker can put whatever source address he wants on an NFS UDP packet.) See the `mountd` or `exports` manual page for more detail. **Note: under no**

circumstances should you make your file system "read-only to the world," as this will let anyone find out NFS file handles. You want the kernel to think of the file system as read-only for the world, but `mountd` to refuse to give out file handles to anybody but '`localhost`'.

'`mountd -n`'.

The `mountd` command takes a flag '`-n`' meaning "allow mount requests from unprivileged ports." **Do not ever run use this flag**. Worse yet, some operating systems (notably HP-UX 9) always exhibit this behavior regardless of whether they '`-n`' flag has been specified.

The '`-n`' option to `mountd` allows any user on an NFS client to learn file handles and thus act as any other user. The situation gets considerably worse when exporting file systems to '`localhost`', however, as SFS requires. Then everybody on the Internet can learn your NFS file handles. The reason is that the `portmap` command will forward mount requests and make them appear to come from '`localhost`'.

`portmap` forwarding

In order to support broadcast RPCs, the `portmap` program will relay RPC requests to the machine it is running on, making them appear to come from '`localhost`'. That can have disastrous consequences in conjunction with '`mountd -n`' as described previously. It can also be used to work around "read-mostly" export options by forwarding NFS requests to the kernel from '`localhost`'.

Operating systems are starting to ship with `portmap` programs that refuse to forward certain RPC calls including mount and NFS requests. Wietse Venema has also written a `portmap` replacement that has these properties, available from `ftp://ftp.porcupine.org/pub/security/index.html`. It is also a good idea to filter TCP and UDP ports 111 (`portmap`) at your firewall, if you have one.

Bugs in the NFS implementation

Many NFS implementations have bugs. Many of those bugs rarely surface when clients and servers with similar implementation talk to each other. Examples of bugs we've found include servers crashing when the receive a write request for an odd number of bytes, clients crashing when they receive the error `NFS3ERR_JUKEBOX`, and clients using uninitialized memory when the server returns a `lookup3resok` data structure with `obj_attributes` having `attributes_follow` set to false.

SFS allows potentially untrusted users to formulate NFS requests (though of course SFS requires file handles to decrypt correctly and stamps the request with the appropriate Unix uid/gid credentials). This may let bad users crash your server's kernel (or worse). Similarly, bad servers may be able to crash a client.

As a precaution, you may want to be careful about exporting any portion of a file system to anonymous users with the '`R`' or '`W`' options to '`Export`' (see [export], page 117). When analyzing your NFS code for security, you should know that even anonymous users can make the following NFS RPC's on a *local-directory* in your '`sfsrwsd_config`' file: `NFSPROC3_GETATTR`, `NFSPROC3_ACCESS`, `NFSPROC3_FSINFO`, and `NFSPROC3_PATHCONF`.

On the client side, a bad, non-root user in collusion with a bad file server can possibly crash or deadlock the machine. Many NFS client implementations have inadequate locking that could lead to race conditions. Other implementations make assumptions about the hierarchical nature of a file system served by the server. By violating these assumptions (for example having two directories on a server each contain the other), a user may be able to deadlock the client and create unkillable processes.

`logger` buffer overrun

SFS pipes log messages through the `logger` program to get them into the system log. SFS can generate arbitrarily long lines. If your `logger` does something stupid like call `gets`, it may suffer a buffer overrun. We assume no one does this, but feel the point is worth mentioning, since not all logger programs come with source.

To avoid using `logger`, you can run `sfscd` and `sfssd` with the '`-d`' flag and redirect standard error wherever you wish manually.

B.7.3 Vulnerabilities in the SFS implementation

Resource exhaustion

The best way to attack the SFS software is probably to cause resource exhaustion. You can try to run SFS out of file descriptors, memory, CPU time, or mount points.

An attacker can run a server out of file descriptors by opening many parallel TCP connections. Such attacks can be detected using the `netstat` command to see who is connecting to SFS (which accepts connections on port 4). Users can run the client (also `sfsauthd`) out of descriptors by connecting many times using the setgid program '`/usr/local/lib/sfs-0.5/suidconnect`'. These attacks can be traced using a tool like lsof, available from `ftp://vic.cc.purdue.edu/pub/tools/unix/lsof`.

SFS enforces a maximum size of just over 64 K on all RPC requests. Nonetheless, a client could connect 1000 times, on each connection send the first 64 K of a slightly larger message, and just sit there. That would obviously consume about 64 Megabytes of memory, as SFS will wait patiently for the rest of the request.

A worse problem is that SFS servers do not currently flow-control clients. Thus, an attacker could make many RPCs but not read the replies, causing the SFS server to buffer arbitrarily much data and run out of memory. (Obviously the server eventually flushes any buffered data when the TCP connection closes.)

Connecting to an SFS server costs the server tens of milliseconds of CPU time. An attacker can try to burn a huge amount of the server's CPU time by connecting to the server many times. The effects of such attacks can be mitigated using hashcash, [HashCost], page 120.

Finally, a user on a client can cause a large number of file systems to be mounted. If the operating system has a limit on the number of mount points, a user could run the client out of mount points.

Non-idempotent operations

If a TCP connection is reset, the SFS client will attempt to reconnect to the server and retransmit whatever RPCs were pending at the time the connection dropped. Not all NFS RPCs are idempotent however. Thus, an attacker who caused a connection to reset at just the right time could, for instance, cause a `mkdir` command to return `EEXIST` when in fact it did just create the directory.

Injecting packets on the loopback interface

SFS exchanges NFS traffic with the local operating system using the loopback interface. An attacker with physical access to the local ethernet may be able to inject arbitrary packets into a machine, including packets to 127.0.0.1. Without packet filtering in place, an attacker can also send packets from anywhere making them appear to come from 127.0.0.1.

On the client, an attacker can forge NFS requests from the kernel to SFS, or forge replies from SFS to the kernel. The SFS client encrypts file handles before giving them to the operating system. Thus, the attacker is unlikely to be able to forge a request from the kernel to SFS that contain a valid file handle. In the other direction however, the reply does not need to contain a file handle. The attacker may well be able to convince the kernel of a forged reply from SFS. The attacker only needs to guess a (possibly quite predictable) 32-bit RPC XID number. Such an attack could result, for example, in a user getting the wrong data when reading a file.

On the server side, you also must assume the attacker cannot guess a valid NFS file handle (otherwise, you already have no security—see [NFS security], page 130). However, the attacker might again forge NFS replies, this time from the kernel to the SFS server software.

To prevent such attacks, if your operating system has IP filtering, it would be a good idea to block any packets either from or to 127.0.0.1 if those packets do not come from the loopback interface. Blocking traffic "from" 127.0.0.1 at your firewall is also a good idea.

Causing deadlock

On BSD-based systems (and possibly others) the buffer reclaiming policy can cause deadlock. When an operation needs a buffer and there are no clean buffers available, the kernel picks some particular dirty buffer and won't let the operation complete until it can get that buffer. This can lead to deadlock in the case that two machines mount each other.

Getting private file data from public workstations

An attacker may be able to read the contents of a private file shortly after you log out of a public workstation if the he can then become root on the workstation. There are two attacks possible.

First, the attacker may be able to read data out of physical memory or from the swap partition of the local disk. File data may still be in memory if the kernel's NFS3 code has cached it in the buffer cache. There may also be fragments of file data in the memory of the `sfsrwcd` process, or out on disk in the swap partition (though `sfsrwcd` does its best to avoid getting paged out). The attacker can read any remaining file contents once he gains control of the machine.

Alternatively, the attacker may have recorded encrypted session traffic between the client and server. Once he gains control of the client machine, he can attach to the `sfsrwcd` process with the debugger and learn the session key if the session is still open. This will let him read the session he recorded in encrypted form.

To minimize the risks of these attacks, you must kill and restart `sfscd` before turning control of a public workstation over to another user. Even this is not guaranteed to fix the problem. It will flush file blocks from the buffer cache by unmounting all file systems, for example, but the contents of those blocks may persist as uninitialized data in buffers sitting on the free list. Similarly, any programs you ran that manipulated private file data may have gotten paged out to disk, and the information may live on after the processes exit.

In conclusion, if you are paranoid, it is best not to use public workstations.

Setuid programs and devices on remote file systems

SFS does its best to disable setuid programs and devices on remote file servers it mounts. However, we have only tested this on operating systems we have access to. When porting SFS to new platforms, It is worth testing that both setuid programs and devices do not work over SFS. Otherwise, any user of an SFS client can become root.

B.8 How to contact people involved with SFS

Please report any bugs you find in SFS to `sfsbug@redlab.lcs.mit.edu`.

You can send mail to the authors of SFS at `sfs-dev@pdos.lcs.mit.edu`.

There is also a mailing list of SFS users and developers at `sfs@sfs.fs.net`. To subscribe to the list, send mail to `sfs-subscribe@sfs.fs.net`.

Concept Index

Bibliography

[1] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, VA, 1993.

[2] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption—how to encrypt with RSA. In A. De Santis, editor, *Advances in Cryptology—Eurocrypt 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1995.

[3] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.

[4] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, Palo Alto, CA, September 1993.

[5] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder. A global authentication service without global trust. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 223–230, Oakland, CA, 1986.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[7] Brent Callaghan and Tom Lyon. The automounter. In *Proceedings of the Winter 1989 USENIX*, pages 43–51. USENIX, 1989.

[8] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylönen. SPKI certificate documentation. Work in progress, from `http://www.pobox.com/~cme/html/spki.html`.

[9] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[10] FIPS 186. *Digital Signature Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, 1994.

[11] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.

[12] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[13] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[14] Kalle Kaukonen and Rodney Thayer. A stream cipher encryption algorithm "arcfour". Internet draft (draft-kaukonen-cipher-arcfour-03), Network Working Group, July 1999. Work in progress.

[15] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer 1990 USENIX*, pages 151–163, Anaheim, CA, 1990. USENIX.

[16] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Network Working Group, November 1998.

[17] Butler Lampson, Martín Abadi, Michael Burrows, and Edward P. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[18] Timothy Mann, Andrew D. Birrell, Andy Hisgen, Chuck Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[19] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[20] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Summer USENIX '90*, pages 247–256, Anaheim, CA, June 1990.

[21] Jan-Simon Pendry. *Amd – an Automounter*. London, SW7 2BZ, UK. Manual comes with amd software distribution.

[22] Niels Provos and David Mazières. A future-adaptable password scheme. In *Proceedings of the 1999 USENIX, Freenix track (the on-line version)*, Monterey, CA, June 1999. USENIX. from `http://www.usenix.org/events/usenix99/provos.html`.

[23] Peter Reiher, Jr. Thomas Page, Gerald J. Popek, Jeff Cook, and Stephen Crocker. Truffles — a secure service for widespread file sharing. In *Proceedings of the PSRG Workshop on Network and Distributed System Security*, pages 101–119, San Diego, CA, 1993.

[24] Ronald L. Rivest and Butler Lampson. SDSI—a simple distributed security infrastructure. Working document from `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[25] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.

[26] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[27] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[28] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[29] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[30] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.

[31] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[32] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988. USENIX.

[33] Amin Vahdat. *Operating System Services for Wide-Area Applications.* PhD thesis, Department of Computer Science, University of California, Berkeley, December 1998.

[34] Hugh C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.

[35] Edward P. Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[36] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[37] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.