17th ACM Symposium on Operating Systems Principles (SOSP '99)
Published as *Operating Systems Review* **34**(5):124–139, Dec. 1999

Separating key management from file system security

David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel
MIT Laboratory for Computer Science
dm@lcs.mit.edu, kaminsky@lcs.mit.edu, kaashoek@lcs.mit.edu, witchel@lcs.mit.edu

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.

1 Introduction

This paper presents SFS, a secure network file system designed to span the Internet. SFS prevents many vulnerabilities caused by today's insecure network file system protocols. It makes file sharing across administrative realms trivial, letting users access files from anywhere and share files with anyone. Most importantly, SFS supports a more diverse range of uses than any other secure file system. It can meet the needs of software vendors, unclassified military networks, and even students running file servers in their dorm rooms. In all cases, SFS strives to avoid cumbersome security procedures that could hinder deployment.

Few people use secure network file systems today, despite the fact that attackers can easily tamper with network traffic. For years, researchers have known how to design and build file systems that work over untrusted networks (for instance Echo [4]). If such a file system could grow to span the Internet, it would let people access and share files securely with anyone anywhere. Unfortunately, no existing file system has realized this goal.

The problem lies in the fact that, at the scale of the Internet, security easily becomes a management and usability nightmare. Specifically, there exists no satisfactory means of managing encryption keys in such a large and diverse network. The wrong key management policy harms security or severely inconveniences people. Yet, on a global scale, different people have vastly different security needs. No single approach to key management can possibly satisfy every user.

Most secure systems limit their usefulness by settling for a particular approach to key management. Consider how few people run secure web servers compared to ordinary ones. Establishing a secure web server with SSL involves significant time, complexity, and cost. Similarly, in the domain of remote login protocols, anyone who has used both Kerberos [29] and the decentralized ssh [34] package knows how poorly the Kerberos security model fits settings in which user accounts are not centrally managed. Unfortunately, most secure file systems come tightly coupled with a key management system that closely resembles either Kerberos or SSL.

SFS takes a new approach to file system security: it removes key management from the file system entirely. SFS introduces *self-certifying pathnames*—file names that effectively contain the appropriate remote server's public key. Because self-certifying pathnames already specify public keys, SFS needs no separate key management machinery to communicate securely with file servers. Thus, while other file systems have specific policies for assigning file names to en-

This research was partially supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288.

SOSP-17 12/1999 Kiawah Island, SC

cryption keys, SFS's key management policy results from the choice users make of which file names to access in the first place.

SFS further decouples user authentication from the file system through a modular architecture. External programs authenticate users with protocols opaque to the file system software itself. These programs communicate with the file system software through well-defined RPC interfaces. Thus, programmers can easily replace them without touching the core of the file system.

Pushing key management out of the file system lets arbitrary key management policies coexist on the same file system, which in turn makes SFS useful in a wide range of file sharing situations. This paper will describe numerous key management techniques built on top of SFS. Two in particular—certification authorities and password authentication—both fill important needs. Neither could have been implemented had the other been wired into the file system.

Without mandating any particular approach to key management, SFS itself also provides a great key management infrastructure. Symbolic links assign human-readable names to self-certifying pathnames. Thus, SFS's global namespace functions as a key certification namespace. One can realize many key management schemes using only simple file utilities. Moreover, people can bootstrap one key management mechanism with another. In practice, we have found the ability to combine various key management schemes quite powerful.

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 [23]) 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 of the authors have their home directories on SFS and perform all their work on it.

2 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.

2.1 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.

2.1.1 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.

2.1.2 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. For most file systems, users must also trust the server to store and return file data correctly (though public, read-only file systems can reside on untrusted servers). 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.

2.1.3 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.

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.

2.2 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 [8], 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 1 shows the format of an actual self-certifying pathname. All remote files in SFS lie under the directory `/sfs`.

[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.

$$/\texttt{sfs}/\overbrace{\texttt{sfs.lcs.mit.edu}}^{\textit{Location}}:\overbrace{\texttt{vefvsv5wd4hz9isc3rb2x648ish742hy}}^{\textit{HostID}\text{ (specifies public key)}}/\overbrace{\texttt{pub/links/sfscvs}}^{\text{path on remote server}}$$

Figure 1: A self-certifying pathname

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 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.

2.3 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 `/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.

2.4 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

[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.

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 *authserv*, use the SRP protocol [33] 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 *authserv* daemon running on his file server. *sfskey* then uses the password as input to SRP to establish a secure channel to the *authserv*. 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 2.5), each user has his own public keys with which to authenticate himself. A users can additionally register an encrypted copies of his private keys with *authserv* 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 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 2.6, 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

[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.

/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 [10] certificates to authenticate SFS servers, as SSL's certification model suits some purposes well. One can in fact build an agent that generates 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.

2.5 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.

2.5.1 *sfsagent* and *authserv*

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, *authserv*.

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 *authserv* validate it. *authserv* maintains a database mapping public keys to user credentials. When it receives a valid request from the file server, *authserv* 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 [34] 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 the request arrive at the agent. Thus, an SFS agent can keep a full audit trail of every private key operation it performs.

2.5.2 User key management

authserv 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. *authserv* 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 *authserv*'s public key databases is configured as either read-only or writable. *authserv* 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 2.4. To ease the adoption of SFS, *authserv* 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 [19]. 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. *authserv* 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 *authserv*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.

2.6 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:

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

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.

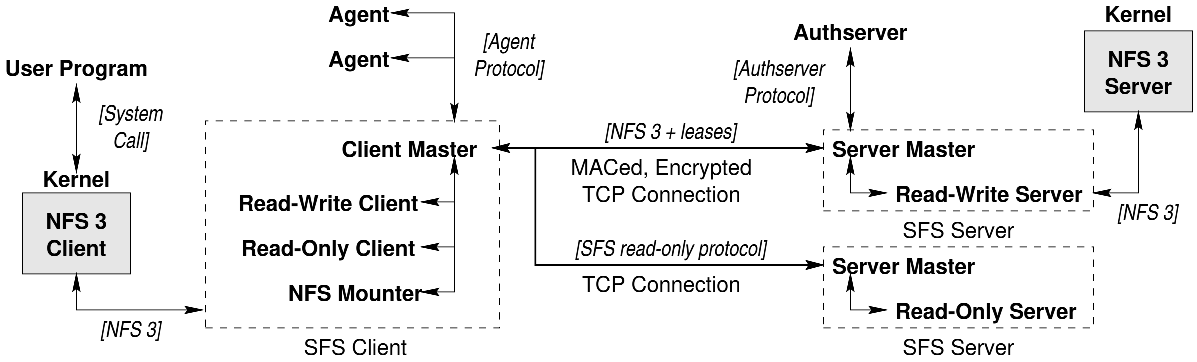

Figure 2: The SFS system components

3 Implementation

Figure 2 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.

3.1 Cryptographic 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}.

3.1.1 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 3, 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 [32]. To ensure forward secrecy, the client employs a short-lived public key, K_C (Figure 3, 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 3.

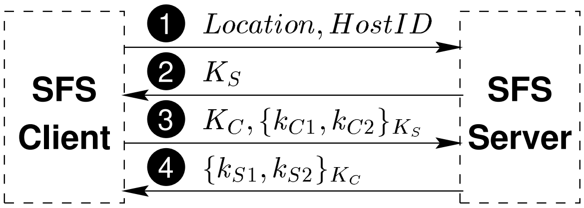

Figure 3: The SFS key negotiation protocol

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).

3.1.2 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:

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

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). 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.[4] If everything succeeds, the 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.

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.

[4] The server accepts out-of-order sequence numbers within a reasonable window to accommodate the possibility of multiple agents on the client returning out of order.

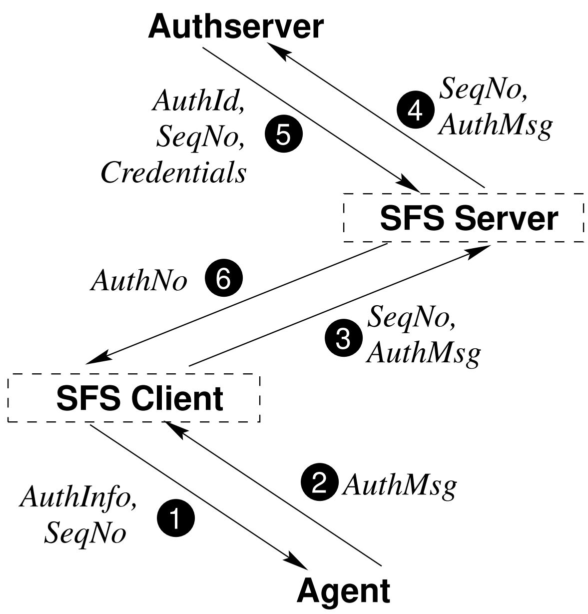

Figure 4: The SFS user authentication protocol

3.1.3 Cryptography

SFS makes three computational hardness assumptions. It assumes the ARC4 [13] 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 [9], 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 [31] 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 guarante the integrity of all file system traffic between clients and read-write servers, and encrypts this trafic 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.

3.2 Modularity and extensibility

Figure 2 reveals that a number of programs comprise the SFS system. All programs communicate with Sun RPC [27]. Thus, the exact bytes exchanged between programs are clearly and unambiguously described in the XDR protocol description language [28]. 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 connections 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. As described in Section 2.4, 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.

3.3 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.

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 [26] 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.

4 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.

4.1 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. However, we believe FreeBSD's TCP implementation of NFS may be suboptimal (in part because we experienced a kernel panic while writing a large file). We therefore mostly consider the comparison between SFS and NFS 3 over UDP. SFS uses UDP for NFS traffic to the local operating system and so is unaffected by bugs in FreeBSD's TCP NFS.

4.2 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.

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 5: Micro-benchmarks for basic operations.

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 5. 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 5. SFS pays 3 Mbyte/sec for its user-level implementation and a further 2.2 Mbyte/sec for encryption.

Although SFS pays a substantial cost for its user-level implementation and software 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 cap file system throughput. Finally SFS's enhanced caching improves performance by reducing the number of RPCs than need to travel over the network.

4.3 End-to-end performance

We evaluate SFS's application performance with the Modified Andrew Benchmark (MAB) [18]. 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.

Figure 6 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

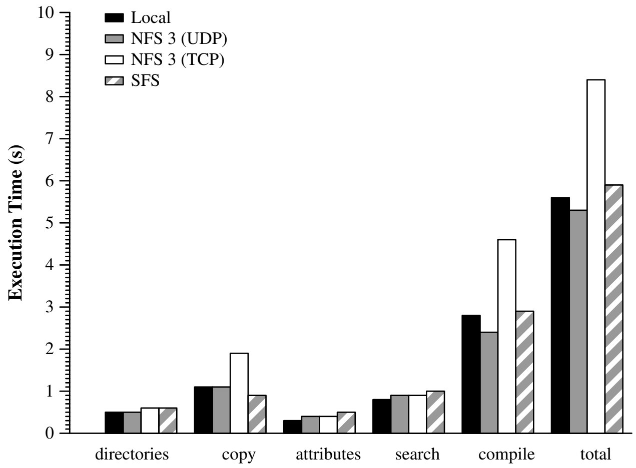

Figure 6: Wall clock execution time (in seconds) for the different phases of the modified Andrew benchmark, run on different file systems. Local is FreeBSD's local FFS file system on the server.

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

Figure 7: Compiling the GENERIC FreeBSD 3.3 kernel.

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 7. 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%.

4.4 Sprite LFS microbenchmarks

The small file test of the Sprite LFS microbenchmarks [22] 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 ran-

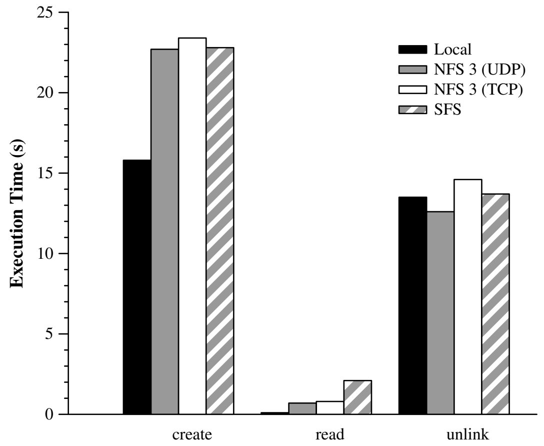

Figure 8: Wall clock execution time for the different phases of the Sprite LFS small file benchmark, run over different file systems. The benchmark creates, reads, and unlinks 1,000 1 Kbyte files. Local is FreeBSD's local FFS file system on the server.

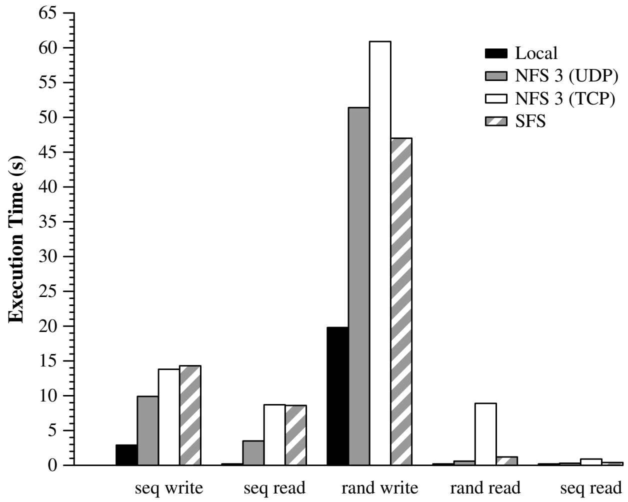

Figure 9: Wall clock execution time for the different phases of the Sprite LFS large file benchmarks, run over different file systems. The benchmark creates a 40,000 Kbyte file and reads and writes 8 Kbyte chunks. Local is FreeBSD's local FFS file system on the server.

domly, and finally 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 8). 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.

4.5 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. 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. We expect this trend to continue.

5 Related work

SFS is the first file system to separate key management from file system security. No other file system has self-certifying pathnames or lets the file namespace double as a key certification 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.

5.1 File systems

AFS [12, 24, 25] 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. Like SFS, AFS mounts all remote file systems under a single directory, `/afs`. 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 [29] 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 [14] 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.[5] 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. 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 *HostID*. They will end up accessing different files with different names, which the file system will consequently cache separately.

Other file systems. The Echo distributed file system [4, 5, 16, 17] 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, much as SFS uses both read-write and read-only file protocols.

The Truffles service [20] is an extension of the Ficus file system [11] 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 [30] 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.

5.2 Internet network security

SSL. SSL [10] 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 SFS or 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

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

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; it lets high- and low-grade certification schemes exist side-by-side. A user can access sensitive servers through `/verisign` without losing the ability to browse sites under `/bargain-cert`. 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 2.4, SFS agents could actually exploit the existing SSL public key infrastructure to authenticate SFS servers.

IPsec. IPsec [15] 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 [7, 21] 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.

6 Summary

SFS requires no information other than a self-certifying pathname to connect securely to a remote file server. As a result, SFS provides a secure, global file system without mandating any particular key-management policy. Other secure file systems all rely on specific policies to assign file names to encryption keys. SFS, in contrast, lets users perform key management by generating file names. In this paper we described many useful key management techniques for SFS that could not have coexisted inside a file system.

Because it has a secure, global namespace, SFS itself constitutes a very effective key management infrastructure. Public keys name files as part of self-certifying pathnames, and files name public keys with symbolic links. Each step of the file name resolution process can invoke a different key management mechanism. The ability to combine multiple mechanisms results in functionality that no one of them can provide alone.

We think that cumbersome security procedures have prevented previous secure file systems from gaining widespread use. We hope SFS will let many people enjoy secure file sharing without an unnecessary administrative burden. To facilitate its deployment, we have made SFS free software.

Acknowledgments

We would like to thank Chuck Blake and Kevin Fu for their contributions to the SFS software. We also thank Chuck for his emergency assistance with hardware, and Kevin for getting us a Verisign certificate and exploring the process. We thank Robert Morris for his help in analyzing various performance artifacts of NFS. We are grateful to our shepherd David Black for the many suggestions on the design and the presentation of SFS. Butler Lampson, David Presotto, Ron Rivest, and the members of PDOS provided insightful comments on this paper.

References

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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`.

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] 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.

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

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

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.