Fast and secure distributed read-only file system

Kevin Fu, M. Frans Kaashoek, David Mazières
{fubob, kaashoek}@lcs.mit.edu, dm@cs.nyu.edu
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
http://www.fs.net/

Abstract

Internet users increasingly rely on publicly available data for everything from software installation to investment decisions. Unfortunately, the vast majority of public content on the Internet comes with no integrity or authenticity guarantees. This paper presents the self-certifying read-only file system, a content distribution system providing secure, scalable access to public, read-only data.

The read-only file system makes the security of published content independent from that of the distribution infrastructure. In a secure area (perhaps off-line), a publisher creates a digitally-signed database out of a file system's contents. The publisher then replicates the database on untrusted content-distribution servers, allowing for high availability. The read-only file system protocol furthermore pushes the cryptographic cost of content verification entirely onto clients, allowing servers to scale to a large number of clients. Measurements of an implementation show that an individual server running on a 550 Mhz Pentium III with FreeBSD can support 1,012 connections per second and 300 concurrent clients compiling a large software package.

1 Introduction

This paper presents the design and implementation of a distributed file system that allows a large number of clients to access public, read-only data securely. Read-only data can have high performance, availability, and security needs. Some examples include executable binaries, popular software distributions, bindings from hostnames to addresses or 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 the replica's own administrator to maliciously serve modified data.

People have introduced a number of ad hoc mechanisms for dealing with the security of public data, but these mechanisms often prove incomplete and of limited utility to other applications. For instance, binary distributions of Linux software packages in RPM [28] format can contain PGP signatures. However, few people actually check these signatures, and packages cannot be revoked. In addition, when packages depend on other packages being installed first, the dependencies cannot be made secure (e.g., one package cannot explicitly require another package to be signed by the same author). As another example, names of servers are typically bound to public keys through digitally signed certificates issued by a trusted authority. These certificates are distributed by the servers they authenticate, which naturally allows scaling to large numbers of servers. However, this approach also results in certificates having a long duration, which complicates revocation to the point that in practice many systems omit it.

To distribute public, read-only data securely, we have built a high-performance, secure, read-only file system designed to be widely replicated on untrusted servers. We chose to build a file system because of

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

the ease with which one can refer to the file namespace in almost any context—from shell scripts to C code to a Web browser's location field. Thus, the file system can support a wide range of applications, such as certificate authorities, that one could not ordinarily implement using a network file system.

Each read-only file system has a public key associated with it. We use the naming scheme of SFS [17], in which file names contain public keys. Thus, users can employ any of SFS's various key management techniques to obtain the public keys of file systems.

In our approach, an administrator creates a database of a file system's contents and digitally signs it off-line using the file system's private key. The administrator then widely replicates the database on untrusted machines. There, a simple and efficient server program serves the contents of the database to clients, without needing access to the file system's private key. DNS round-robin scheduling or more advanced techniques can be used to distribute the load between multiple replicas. A trusted program on the client machine checks the authenticity of data before returning it to the user.

The read-only file system avoids performing any cryptographic operations on servers and keeps the overhead of cryptography low on clients. We accomplish this with two simple techniques. First, blocks and inodes are named by *handles*, which are collision-resistant cryptographic hashes of their contents. Second, groups of handles are hashed recursively, producing a tree of hashes. Inodes contain the handles of a file's blocks. Directory blocks contain lists of file name to handle bindings. Using the handle of the root inode of a file system, a client can verify the contents of any block by recursively checking hashes.

The protocol between the client and server consists of only two remote procedure calls: one to fetch the signed handle for the root inode of a file system, and one to fetch the data (inode or file content) for a given handle. Since the server does not have to understand what it is serving, its implementation is both trivial and highly efficient: it simply looks up handles in the database and sends them back to the client.

We named the file system presented in this paper the *SFS read-only file system* because it uses SFS's naming scheme and fits into the SFS framework. The server-side of the file system consists of two programs: `sfsrodb` for creating signed databases offline, and `sfsrosd` for serving signed databases from untrusted machines. The client-side software consists of a daemon, `sfsrocd`, that queries databases through `sfsrosd` processes, verifies the results, and translates them into a file system.

A performance evaluation shows that `sfsrosd` can support 1,012 short-lived connections per second on a PC (a 550 Mhz Pentium III with 256 Mbyte of memory) running FreeBSD, which is 26 times better than a standard read-write SFS file server and 92 times better than a secure web server. In fact, the performance of the read-only server is limited mostly by the number of TCP connections per second, not by the overhead of cryptography, which is offloaded to clients. For applications like sustained downloads that require longer-lived connections, `sfsrosd` can support 300 concurrent sessions while still saturating a fast Ethernet.

The rest of this paper is organized as follows. Section 2 relates our design to previous work. Section 3 details the design of the read-only server. Section 4 describes its implementation. Section 5 presents the applications of the read-only server. Section 6 evaluates the performance of these application and compares them to existing approaches. Section 7 concludes.

2 Related Work

We are unaware of a read-only (or read-write) file system that can support a high number of simultaneous clients and provide strong security. Many sites use a separate file system to replicate and export read-only binaries, providing high availability and high performance. AFS supports read-only volumes to achieve replication [22]. However, in all these cases replicas are stored on trusted servers. Some file systems provide high security (e.g., the SFS read-write file system [17] or Echo [2]), but compared to the SFS read-only file system these servers do not scale well with the number of clients because their servers perform expensive cryptographic operations in the critical path (e.g., the SFS read-write server performs one private-key operation per client connection, which takes about 24 msec on a 550 Mhz Pentium III).

Secure DNS [8] is an example of a read-only data service that provides security, high availability, and high performance. In secure DNS, each individual resource record is signed. This approach does not work for file systems. If each inode and 8 Kbyte-block of a moderate file system—for instance, the 635 Mbyte Red Hat 6.2 i386 distribution—had to be signed individually with a 1,024-bit key, the signing alone would take about 36 minutes ($90,000 \times$ 24 msec) on a 550 Mhz Pentium III. A number of read-only data services, such as FTP archives, are 100 times bigger, making individual block signing impractical—particularly since we want to allow frequent updates of the database and rapid expiration of old signatures.

Secure HTTP servers are another example of servers that provide access to mostly read-only data. These servers are difficult to replicate on untrusted machines, however, since their private keys have to be on-line to prove their identity to clients. Furthermore, private-key operations are expensive and are in the critical path: every SSL connection requires the server to compute modular exponentiations as part of the public-key cryptography [11]. As a result, software-only secure Web servers achieve low throughput (with a 1,024-bit key, IE and Netscape servers can typically support around 15 connections per second).

Content distribution networks built by companies such as Adero, Akamai, Cisco, and Digital Island are an efficient and highly-available way of distributing static Web content. Content stored on these networks is dynamically replicated on trusted caches scattered around the Internet. Web browsers then connect to a cache that provides high performance. The approach described in this paper would allow read-only Web content to be replicated securely to *untrusted* machines and would provide strong data integrity to clients that run our software. For clients that don't run our software, one can easily configure any Web server on an SFS client to serve the `/sfs` directory, trivially creating a Web-to-SFS gateway for any Web clients that trust the server.

Signed software distributions are common in the open-source community. In the Linux community, for example, a creator or distributor of a software package can sign RPM [28] files with PGP or GNU GPG. RPM also supports MD5 hashes. A person downloading the software can optionally check the signature. Red Hat Software, for example, publishes their PGP public key on their Web site and signs all their software distributions with the corresponding private key. This setup provides some guarantees to the person who checks the signature on the RPM file and who makes sure that the public key indeed belongs to Red Hat. However, RPMs do not provide an expiration time or revocation support. If users were running the SFS client software and RPMs were stored on SFS read-only file servers, the server would be authenticated transparently and the data would be checked transparently for integrity and recentness.

The read-only file system makes extensive use of hash trees, which have appeared in numerous other systems. Merkle used a hierarchy of hashes for an efficient digital signature scheme [18]. In the context of file systems, the Byzantine-fault-tolerant file system uses hierarchical hashes for efficient state transfers between clients and replicas [5, 6]. The cryptographic storage file system [12] uses cryptographic hashes in a similar fashion to the SFS read-only file system. Duchamp uses hierarchical hashes to efficiently compare two file systems in a toolkit for partially-connected operation [7]. TDB [16] uses hash trees combined with a small amount of trusted storages to construct a trusted database system on untrusted storage. Finally, a version of a network-attached storage device uses an incremental "Hash and MAC" scheme to reduce the cost of protecting the integrity of read traffic in storage devices that are unable to generate a MAC at full data transfer rates [14].

A number of proposals have been developed to make digital signatures cheaper to compute [13, 20], some involving hash trees [27]. These proposal enable signing hundreds of packets per second in applications such as multicast streams. However, if applied to file systems, these techniques would introduce complications such as increased signature size. Moreover, because the SFS read-only file system was designed to avoid trusting servers, read-only servers must function without access to a file system's private key. This prevents any use of dynamically computed digital signatures, regardless of the computational cost.

3 SFS read-only file system

Figure 1 shows the overall architecture of the SFS read-only file system. In a secure area, an administrator runs the SFS read-only database generator (`sfsrodb`), passing as arguments a directory of files to export and a file containing a private key. The administrator replicates this database on a number of untrusted machines, each of which runs a copy of the SFS read-only server daemon (`sfsrosd`). The

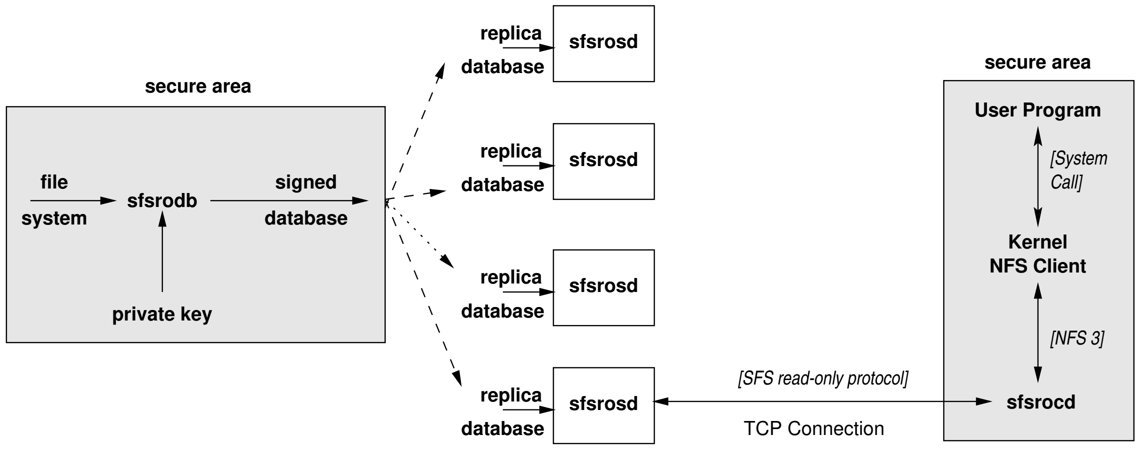

Figure 1: The SFS read-only file system. Shaded boxes show the trusted computing base.

read-only server is a simple program that looks up the data for a given handle in the replica's database and returns the data to the client.

Most of the actual file system is implemented by the SFS read-only client daemon (`sfsrocd`), which runs on the client's machine. It handles file system requests from the local operating system and responds to them. The read-only client understands the format of inodes and directories. It parses pathnames, searches directories, looks up blocks of files, etc.

In order to respond to requests from the local operating system, the client retrieves data from one of the servers that has a replica of the database. DNS round-robin scheduling or more advanced techniques (e.g., [15]) can be used to select a replica that provides good performance. Since the replica may run on untrusted hardware, the client must verify that any data sent by the server was indeed signed by a database generator with the appropriate private key.

The SFS read-only file system assumes that an attacker may compromise and assume control of any read-only server machine. It therefore cannot prevent denial-of-service from an attacker penetrating and shutting down every server for a given file system. However the client does ensure that any data retrieved from a server is authentic, no older than a file system-configurable consistency period, and also no older than any previously retrieved data from the same file system. The read-only file system does not provide confidentiality. Thus, data on replicas does not have to be kept secret from attackers. The key security property of the read-only file system is integrity.

Operation	Cost (μsec)
Sign 68 byte fsinfo	24,400
Verify 68 byte fsinfo	82
SHA-1 256 byte iv+inode	17
SHA-1 8,208 byte iv+block	406

Figure 2: Performance of base primitives on a 550 Mhz Pentium III. Signing and verification use 1,024-bit Rabin-Williams keys.

Our design can also in principle be used to provide non-repudiation of file system contents. An administrator of a server could commit to keeping every file system he ever signed. Then, clients could just record the signed root handle. The server would be required to prove what the file system contained on any previous day. In this way, an administrator could never falsely deny that a file previously existed.

Figure 2 lists the cryptographic primitives that we use in the read-only file system. We chose the Rabin public key cryptosystem [26] for its fast signature verification time. The implementation is secure against chosen-message attacks (using the redundancy function proposed in [1]). As can be seen from Table 2, computing digital signatures is somewhat expensive, but verifying them is takes only 82 μsec—far cheaper than a typical network round trip time, in fact.

SFS also uses the SHA-1 [9] cryptographic hash function. SHA-1 is a collision-resistant hash function that produces a 20-byte output from an arbitrary-length input. Finding any two inputs of SHA-1 that

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

Figure 3: Contents of the digitally signed root of an SFS read-only file system.

produce the same output is believed to be computationally intractable. Modern machines can typically compute SHA-1 at a rate greater than the local area network bandwidth. Thus, one can reasonably hash the result of every RPC in a network file system protocol.

The rest of this section describes how we use these primitives to efficiently provide authenticity and recentness of data.

3.1 SFS read-only Protocol

The read-only protocol uses two RPCs: *getfsinfo* and *getdata*. *getfsinfo* takes no arguments and returns a digitally signed `FSINFO` structure, depicted in Figure 3. The SFS client verifies the signature using the public key embedded in the server's name. The *getdata* RPC takes a 20-byte 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 `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 `FSINFO` indicates the time (in seconds since 1970) at which a file system was signed. Clients cache 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 `FSINFO` structure when the current time exceeds `start` + `duration`.

The file system names arbitrary-length blocks of data with fixed-size handles. The handle for a data item x is computed using SHA-1: $H(x) = \text{SHA-1}(\texttt{iv}, x)$. `iv`, the initialization vector, is randomly chosen by *sfsrodb* the first time the administrator 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 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, as described below. `fhdb` is the the hash of the root of a tree that contains every handle reachable from the root directory. `fhdb` lets clients securely verify that a particular handle does not exist, so that they can return stale file handle errors when file systems change. `fhdb` will not be necessary in future versions of the software, as described in Section 3.4.

3.2 SFS read-only inode

Figure 4 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, opaque directory, or symbolic link), size, and modification time. Permissions are not included because they can be synthesized on the client. The inode then contains handles of successive 8 Kbyte blocks of file data. If the file contains more than eight blocks, the inode contains the handle of an *indirect block*, which in turn contains handles of file blocks. Similarly, for larger files, an inode can also contain the handles of double- and triple-indirect blocks. In this way, the blocks of small files can be verified directly from the inode, while inodes can also indirectly verify large files—an approach similar to the on-disk data structures of the Unix File System [19].

3.3 Database generator

To export a file system, a system administrator produces a signed database from a source directory in an existing file system. The database contains file data blocks and inodes indexed by their hash values. In essence, it is analogous to a file system in which inode and block numbers have been replaced by cryptographic hashes.

The database generator utility traverses the given file system depth-first to builds the database. The

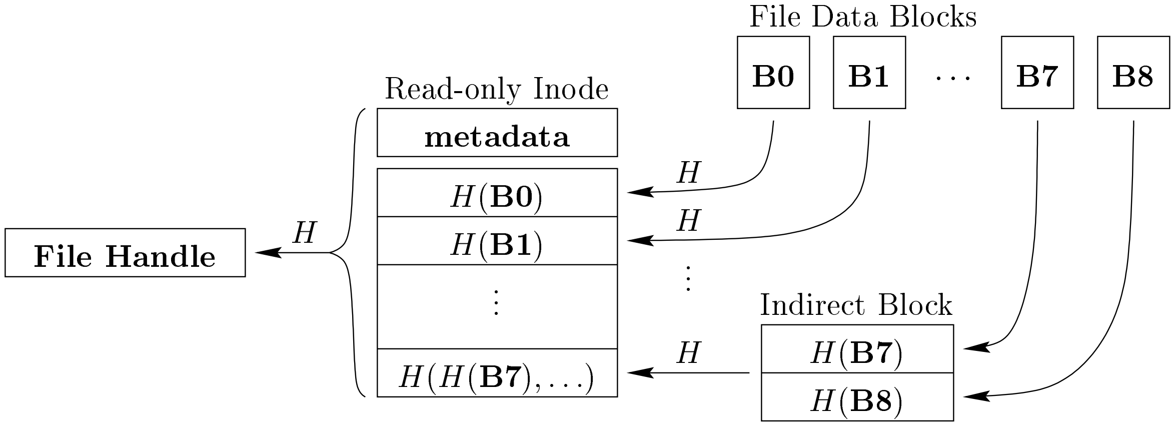

Figure 4: Format of a read-only file system inode.

leaves of the file system tree are files or symbolic links. For each regular file in a directory, the database generator creates a read-only inode structure and fills in the metadata. Then, it reads the blocks of the file. For each block, `sfsrodb` hashes the data in that block to compute its handle, and then inserts the block into the database under the handle (i.e., a lookup on the handle will return the block). The hash value is also stored in an inode. When all file blocks of a file are inserted into the database, the filled-out inode is inserted into the database under its hash value.

When all files in a given directory have been inserted into the database, the generator utility inserts a file corresponding to the directory itself. The file blocks of a directory contain lists of $\langle$name, handle$\rangle$ pairs; the directory's inode contains hashes of those blocks (and possibly indirect blocks) as for regular files. Directories are sorted lexicographically by name. Thus, clients can avoid traversing the entire directory by performing a binary search when looking up files in very large directories (e.g., a directory that contains all names in the `.com` domain). This property also allows clients to verify inexpensively whether a file name exists or not, without having to read the whole directory.

Each directory also contains its full pathname from the root of the file system. The client uses the pathname to evaluate the file name "`..`" locally, using it as a reference for any directory's parent. (Since a directory inode's handle depends on the handles of all subdirectories, a circular dependency makes it impossible to create directory entries of the form $\langle$"`..`", parent's handle$\rangle$.) Clients verify that a directory contains the proper pathname when first looking it up. This is not strictly necessary—an administrator signing a bad database should expect undefined interpretations by clients. However, the sanity check reduces potentially confusing behavior on clients of malformed file systems.

Inodes for symbolic links are slightly different from the one depicted in Figure 4. Instead of containing handles of blocks, the inode directly contains the destination path for the symbolic link.

To avoid inconveniencing users with large directories, server administrators can set the type field in an inode to "opaque directory." When users list an opaque directory, they see only entries they have already referenced—somewhat like Unix "automounter" directories [4]. Opaque directories are well-suited to giant directories containing, for instance, all names in the `.com` domain or all name-to-key bindings issued by a particular certificate authority. If one used non-opaque directories for these applications, users could inadvertently download hundreds of megabytes of directory data by typing `ls` or using file name completion in the wrong directory.

After the whole directory tree has been inserted into the database, the generator utility fills out an `FSINFO` structure and signs it with the private key of the file system. For simplicity, `sfsrodb` stores the signed `FSINFO` structure in the database under a well-known, reserved key.

The database generator stores inodes, directories, and file block data in the database in XDR marshaled form [24]. Using XDR has three advantages. First, it simplifies the client implementation, as the client can use the SFS RPC and crypto libraries to parse file system data. Second, the XDR representation clearly defines what the database contains, which simplifies writing programs that process the database (e.g., a debugging program). Finally, it improves performance of the read-only server by saving it from having to do any marshaling.

3.4 Updating file systems

The biggest challenge in updating read-only file systems is dealing with data that no longer exists in the file system. When a file system changes, the administrator generates a new database and pushes it out to the server replicas. Files that persist across file system versions will keep the same handles. However, when a file is removed or modified, clients can end up requesting handles no longer in the database. In this case, the read-only server replies with an error.

Unfortunately, since read-only servers (and the network) are not trusted, clients cannot necessarily believe "handle not found" errors they receive. Though a compromised server can hang a client by refusing to answer RPCs, it must not be able to make programs spuriously abort with stale file handle errors. Otherwise, for instance, an application looking up a key revocation certificate in a read-only file system might falsely believe that the certificate did not exist.

We have two schemes to let clients securely determine whether a given file handle exists: the current scheme uses the `fhdb` field of the `FSINFO` structure to verify that a handle no longer exists. `fhdb` is the root of a hash tree, the leaf nodes of which contain a sorted list of every handle in the file system. Thus, clients can easily walk the hash tree (using *getfsinfo*) to see whether the database contains a given file handle.

The `fhdb` scheme has advantages. It allows files to persist in the database even after they have been deleted, as not every handle in the database need be reachable from the root directory. Thus, by keeping handles of deleted files in a few subsequent revisions of a database, a system administrator can support the traditional Unix semantics that one can continue to access an open file even after it has been deleted.

Unfortunately, `fhdb` has several drawbacks. Even small changes to the file system cause most of the hash tree under `fhdb` to change (making incremental database updates unnecessarily expensive). Furthermore, in the read-only file system, because handles are based on file contents, there is no distinction between modifying a file and deleting then recreating it. In some situations, one doesn't want to have to close and reopen a file to see changes. (This is always the case for directories, which therefore need a different mechanism anyway.) Finally, under the `fhdb` scheme, a server cannot change its `iv` without causing all open files to become stale on all clients.

To avoid these problems, future versions of the software will eliminate `fhdb`. Instead, the client will track the pathnames of all files accessed in read-only file systems. When a server `FSINFO` structure is updated, the client will walk the file namespace to find the new inode corresponding to the name of each open file. Those who really want an open file never to change can still emulate the old semantics (albeit somewhat inelegantly) using a symbolic link to switch between the old and new version of a file while allowing both to exist simultaneously. Once clients track the pathnames of files, directories need no longer contain their full pathnames: clients will have enough state to evaluate the parent directory name "`..`" on their own.

The read-only inode structure contains the modification and "inode change" times of a file. Thus, `sfsrodb` could potentially update the database incrementally after changes are made to the file system, recomputing only the hashes from changed files up to the root handle and the signature on the `FSINFO` structure. Our current implementation of `sfsrodb` creates a completely new database for each version of the file system, but we plan to support incremental updates in a future release.

3.5 Incremental transfer

We built a simple utility program, `pulldb`, that incrementally transfers a newer version of a database from one replica to another. The program fetches `FSINFO` from the source replica, and checks if the local copy of the database is out of date. If so, the program recursively traverses the entire file system, starting from the new root file handle, building on the side a list of all active handles. For each handle encountered, if the handle does not already exist in the local database, `pulldb` fetches the corresponding data with a *getdata* RPC and stores it in database. After the traversal, `pulldb` swaps the `FSINFO` structure in the database and then deletes all handles no longer in the file system. If a failure appears before the transfer is completed, the program can just be restarted, since the whole operation is idempotent.

3.6 Read-only server

The server program `sfsrosd` is a trivial—only 400 lines of C++. `sfsrosd` knows nothing about the structure of the file system it serves; it simply gets requests for handles, looks up the handles in the database, and returns their values to the client. It also fields *getfsinfo* and SFS *connect* RPCs, to which it replies with two static data structures cached in memory.

3.7 Read-only client

The client program constitutes the bulk of the code in the read-only file system (1,500 lines of C++). The read-only client behaves like an NFS3 [3] server, allowing it to communicate with the operating system through ordinary networking system calls. The read-only client resolves pathnames for file name lookups and handles reads of files, directories, and symbolic links. It relies on the server only for serving blocks of data, not for interpreting or verifying those blocks. The client checks the validity of all blocks it receives against the hashes by which it requested them.

We demonstrate how the client works by example. Consider a user reading the file `/sfs/sfs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69/README`, where `bzcc5hder7cuc86kf6qswyx6yuemnw69` is the representation of the public key of the server storing the file `README`. (In practice, symbolic links save users from ever having to see or type pathnames like this.)

The local operating system's NFS client will call into the protocol-independent SFS client software, asking for the directory `/sfs/sfs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69/`. The client will contact `sfs.mit.edu`, which will respond that it implements the read-only file system protocol. At that point, the protocol-independent SFS client daemon will pass the connection off to the read-only client, which will subsequently be asked by the kernel to interpret the file named `README`.

The client makes a *getfsinfo* RPC to the server to get the file system's signed `FSINFO` structure. It verifies the signature on the structure, ensures that the `start` field is no older than its previous value if the client has seen this file system before, and ensures that `start` + `duration` is in the future.

The client then obtains the root directory's inode by doing a *getdata* RPC on the `rootfh` field of `FSINFO`. Given that inode, it looks up the file `README` by doing a binary search among the blocks of the directory, which it retrieves through *getdata* calls on the block handles in the directory's inode (and possibly indirect blocks). When the client has the directory entry $\langle$`README`, handle$\rangle$, it calls *getdata* on handle to obtain `README`'s inode. Finally, the client can retrieve the contents of `README` by calling *getdata* on the block handles in its inode.

4 Implementation

As illustrated in Figure 5, the read-only file system is implemented as two new daemons (`sfsrocd` and `sfsrosd`) in the SFS system [17]. `sfsrodb` is a stand-alone program.

`sfsrocd` and `sfsrosd` communicate with Sun RPC over a TCP connection. (The exact message formats are described in the XDR protocol description language [24].) We also use XDR to define cryptographic operations. Any data that the read-only file system hashes or signs is defined as an XDR data structure; SFS computes the hash or signature on the raw, marshaled bytes.

`sfsrocd`, `sfsrosd`, and `sfsrodb` are written in C++. To handle many connections simultaneously, the client and server use SFS's asynchronous RPC library. Both programs are single-threaded, but the RPC library allows the client to have many outstanding RPCs.

Because of SFS's support for developing new servers and `sfsrosd`'s simplicity, the implementation of `sfsrosd` is trivial. It gets requests for data blocks by file handle, looks up preformatted responses in a B-tree, and responds to the client. The current implementation uses the Sleepycat database's B-tree [23]. In the measurements `sfsrosd` accesses the database synchronously.

The implementations of the other two programs (`sfsrodb` and `sfsrocd`) are more interesting; we discuss them in more detail.

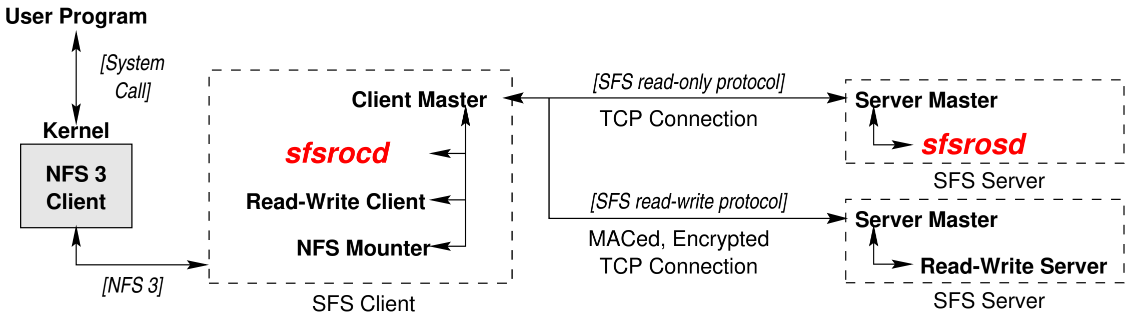

Figure 5: Implementation overview of the read-only file system in the SFS framework.

4.1 sfsrodb

The implementation of `sfsrodb` is simple. It is a short stand-alone C++ program. It walks down the given file system and builds a database with file system data indexed by the cryptographic hash of the data. After building the database, it creates an `FSINFO` structure and signs it with the private key of the file system.

The disadvantage of storing the data in marshaled form is that physical representation of the data is slightly larger than the actual data. For instance, an 8 Kbyte file block is slightly larger than 8 Kbyte. The Sleepycat database does not support values just over a power of 2 in size very well; we are developing a light-weight, asynchronous B-tree that handles such odd-sized values well.

A benefit of storing blocks under their hash is that blocks from different files that have the same hash will only be stored once in the database. If a file system contains blocks with identical content among multiple files, then `sfsrodb` stores just one block under the hash. In the RedHat 6.2 distribution, 5,253 out of 80,508 file data blocks share their hash with another block. The overlap is much greater if one makes the same data available in two different formats (for instance, the contents of the RedHat 6.2 distribution, and the image of a CD-ROM containing that distribution).

4.2 sfsrocd

`sfsrocd` implements four caches with LRU replacement policies to improve performance by avoiding RPCs to `sfsrosd`. It maintains an inode cache, an indirect-block cache, a small file-block cache, and a cache for directory entries.

`sfsrocd`'s small file-block cache primarily optimizes the case of the same block appearing in multiple files. In general, `sfsrocd` relies on the local operating system's buffer cache to cache the file contents. Thus, any additional caching of file contents will tend to waste memory unless a block in appears multiple places. The small block cache optimizes common cases—such as a file with many blocks of all zeros—without dedicating too much memory to redundant caching.

Indirect blocks are cached so that `sfsrocd` can quickly fetch and verify multiple blocks from a large file without refetching the indirect blocks. `sfsrocd` does not prefetch because most operating systems already implement prefetching locally.

5 Applications

To demonstrate the usefulness of the SFS read-only file system, we describe two applications that we measure in Section 6: certificate authorities and software distribution.

5.1 Certificate Authorities

Certificate authorities for the Internet are servers that publish certificates binding hostnames to public keys. On the Web, for instance, the certificate authority Verisign certifies server keys for Web servers. Verisign signs the domain name and the public key of the Web server in an X.509 certificate, and returns this to the Web server administrator [10]. When a browser connects to the Web server with secure HTTP, the server responds with the certificate. The browser checks the validity of the certificate by verifying it with Verisign's public key. Most popular browsers have Verisign's key embedded in their binaries. One benefit of this approach is that Verisign does not have to be on-line when the browser connects to a certified Web server. However, this comes at the cost of complicating certificate revocation to the point that in practice no one does it.

In contrast, SFS uses file systems to certify public keys of servers. SFS certificate authorities are nothing more than ordinary file systems serving symbolic links that translate human-readable names into public keys that name file servers [17]. For example, if Verisign acted as an SFS certificate authority, client administrators would likely create symbolic links from their local disks, for instance `/verisign`, to Verisign's self-certifying pathname—a pathname containing the public key of Verisign's file system. This file system would in turn contain symbolic links to other SFS file systems. For example, `/verisign/NYU` might be a symbolic link to a self-certifying pathname for an SFS file server that Verisign calls NYU.

Unlike traditional certificate authorities, SFS certificate authorities get queried interactively. This simplifies certificate revocation, since revoking a key amounts to removing the symbolic link. However, it also places high integrity, availability, and performance demands on file systems serving as on-line certificate authorities.

By running certificate authorities as SFS read-only file systems, we can address these needs. The SFS read-only file system improves performance by making the amount of cryptographic computation proportional to the file system's size and rate of change, rather than to the number of clients connecting. SFS read-only also improves integrity by freeing SFS certificate authorities from the need to keep any on-line copies of their private keys. Finally, SFS read-only improves availability because it can be replicated on untrusted machines.

An administrator adds certificates to its SFS file system by adding new symbolic links. The database is updated once a day, similarly to second-level DNS updates. The administrator (incrementally) replicates the database to other servers.

The certificate authority database (and thus its certificates) might be valid for one day. The certificate that we bought from Verisign for our Web server is valid for 12 months. If the private key of an SFS server is compromised, then the next day the certificate will be out of the on-line database.

SFS certificate authorities also support key revocation certificates to revoke public keys of servers explicitly. The key revocation certificates are self-authenticating [17] and signed with the private key of the compromised server. Verisign could, for example, maintain an SFS certificate authority that has a directory to which users upload revocation certificates for some fee; since the certificates are self-authenticating, Verisign does not have to certify them. Clients check this directory when they perform an on-line check for key certificates. Because checks can be performed interactively, this approach works better than X.509 certificate revocation lists [10].

5.2 Software Distribution

Sites distributing popular software have high availability, integrity, and performance needs. Open software is often replicated at several mirrors to support a high number of concurrent downloads. If users download a distribution with anonymous FTP, they have low data integrity: a user cannot tell whether he is downloading a trojan-horse version instead of the correct one. If users connect through the Secure Shell (SSH) or secure HTTP, then the server's throughput is low because of cryptographic operations it must perform. Furthermore, that solution doesn't protect against attacks where the server is compromised and the attacker replaces a program on the server's disk with a trojan horse.

By distributing software through SFS read-only servers, one can provide integrity, performance, and high availability. Users with `sfsrocd` can even browse the distribution as a regular file system and compile the software straight from the sources stored on the SFS file system. `sfsrocd` will transparently check the authenticity of the file system data. To distribute new versions of the software, the administrator simply updates the database. Users with only a browser could get all the benefits by just connecting through a Web-to-SFS proxy to the SFS file system.

Software distribution using the read-only file systems complements signed RPMs. First, RPMs do not provide any revocation support; the signature on an RPM is good forever. Second, there is no easy way to determine whether an RPM is recent; an attacker can give a user an older version of a software package without the user knowing it. Third, there is no easy method for signing a collection of RPMs that constitute a single system. For an example, there is currently no way of cryptographically verifying that one has the complete Linux RedHat 6.2 distribution (or all necessary security patches for a release). Using SFS read-only, Red Hat could securely distribute the complete 6.2 release, providing essentially the same security guarantees as a physical CDROM distribution.

6 Performance

This section presents the results of measurements to support the claims that (1) SFS read-only provides acceptable application performance and (2) SFS read-only scales well with the number of clients.

To support the first claim, we measure the performance of microbenchmarks and a large software compilation. We compare the performance of the SFS read-only file system with the performance on the local file system, insecure NFS, and the secure SFS read-write file system.

To support the second claim, we measure the maximum number of connections per server and the throughput of software downloads with an increasing number of clients.

We expect that the main factors affecting SFS read-only performance are the user-level implementation in the client, hash verification in the client, and database lookups on the server.

6.1 Experimental setup

We measured performance on 550 MHz Pentium IIIs running FreeBSD 3.3. The client and server were connected by 100 Mbit, full-duplex, switched Ethernet. Each machine had a 100 Mbit Tulip Ethernet card, 256 Mbytes of memory, and an IBM 18ES 9 Gigabyte SCSI disk. In `sfsrocd`, the inode, indirect-block, and directory entry caches each have a maximum of 512 entries, while the file-block cache has maximum of 64 entries. Maximum TCP throughput between client and server, as measured by `ttcp` [25], was 11.31 Mbyte/sec.

Because the certificate authority benchmark in Section 6.4 requires many CPU cycles on the client, we also employed two 700 MHz Athlons running OpenBSD 2.7. Each Athlon had a 100 Mbit Tulip Ethernet card and 128 Mbytes of memory. Maximum TCP throughput between an Athlon and the FreeBSD server, as measured by `ttcp`, was 11.04 Mbyte/sec. The Athlon machines generated the client SSL and SFSRW requests; we report the sum of the performance measured on the two machines.

For all experiments we report the average of five runs.

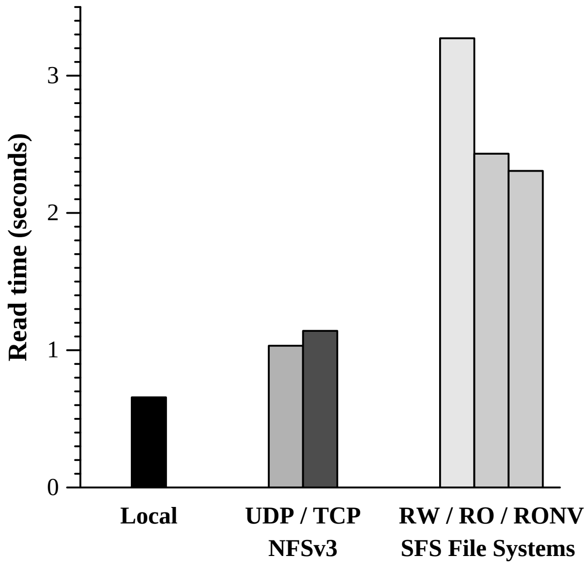

Figure 6: Time to sequentially read 1,000 1 Kbyte files. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warmed server caches, but cold client caches. RW, RO, and RONV denote respectively the read-write protocol, the read-only protocol, and the read-only protocol with no verification.

6.2 Microbenchmarks

To evaluate the performance of the SFS read-only system, we perform small and large file microbenchmarks.

6.2.1 Small file benchmark

We use the read phases of the LFS benchmarks [21] to obtain a basic understanding of single client/single server performance. Figure 6 shows the latency of sequentially reading 1,000 1 Kbyte

Breakdown	Cost (sec)
NFS loopback	0.661
Computation in client	1.386
Communication with server	0.507
Total	2.55

Table 1: Breakdown of SFS read-only performance reported in Fig 6.

files on the different file systems. The files contain random data and are distributed evenly across ten directories. For the read-only and NFS experiments, all samples were within 0.4% of the average. For the read-write experiment, all samples were within 2.7% of the average. For the local file system, all samples were within 6.9% (0.4 seconds) of the average.

As expected, the SFS read-only server performs better than the SFS read-write server (2.43 vs. 3.27 seconds). The read-only file server performs worse than NFSv3 over TCP (2.43 vs. 1.14 seconds). To understand the performance of the read-only file server, we break down the 2.43 seconds spent in the read-only client (see Table 1).

To measure the cost of the user-level implementation we measured the time spent in NFS loopback. We used the `fchown` operation against a file in a read-only file system to measure the time spent in the user-level NFS loopback file system. This operation generates NFS RPCs from the kernel to the read-only client, but no traffic between the client and the server. The average over 1,000 `fchown` operations is 167 μsec. By contrast, the average for attempting an `fchown` of a local file with permission denied is 2.4 μsec. The small file benchmark generates 4,015 NFS loopback RPCs. Hence, the overhead of the client's user-level implementation is at least (167 μsec - 2.4 μsec) * 4,015 = 0.661 seconds.

We also measured the CPU time spent during the small file benchmark in the read-only client at 1.386 seconds. With verification disabled, this drops to 1.300 seconds, indicating that for this workload, file handle verification consumes very little CPU time.

To measure the time spent communicating with the read-only server, we timed the playback of a trace of the 2,101 *getdata* RPCs of the benchmark to the read-only server. This took 0.507 seconds.

These three measurements total to 2.55 seconds. With an error margin of 5%, this accounts for the 2.43 seconds to run the benchmark. We attribute this error to a small amount of double counting of cycles between the NFS loopback measurement and the computation in the client.

The cryptography accounts for very little of the time. The CPU time spent on verification is only 0.086 seconds. Moreover, end-to-end measurements show that data verification has little impact on performance. RONV performs slightly better than RO (2.31 vs. 2.43 seconds). Therefore, any optimization will have to focus on the non-cryptographic portions of the system.

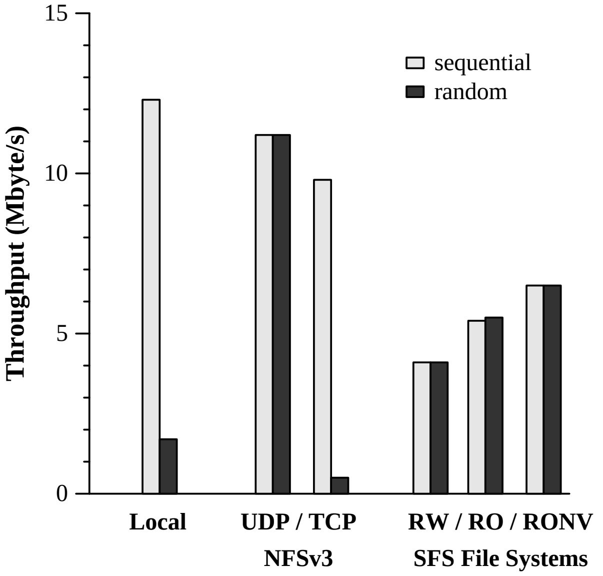

Figure 7: Throughput of sequential and random reads of a 40 Mbyte file. The experimental conditions are the same as in Figure 6.

6.2.2 Large file benchmark

Figure 7 shows the performance of sequentially and randomly reading a large (40 Mbyte) file containing random data. We read in blocks of 8 Kbytes. In the network experiments, the file is in the server's cache, but not in the client's cache. Thus, we are not measuring the server's disk. This isolates the software overhead of cryptography and SFS's user-level design. For the local file system, all samples were within 1.4% of the average. For NFSv3 over UDP and the read-write experiments, all samples were within 1% of the average. For NFSv3 over TCP and the read-only experiments, all samples were within 4.3% of the average. This variability and the poor

NFSv3 over TCP performance appears to be due to a pathology of FreeBSD 3.3.

The SFS read-only server performs better than the read-write server because the read-only server performs no on-line cryptographic operations. On the sequential workload, verification costs 1.4 Mbyte/s in throughput. NFSv3 over TCP performs substantially better (9.8 vs. 6.5 Mbyte/s) than the read-only file system without verification, even though both run over TCP and do similar amounts of work; the main difference is that NFS is implemented in the kernel.

If the large file contains only blocks of zeros, SFS read-only obtains a throughput of 17 Mbyte/s since all blocks hash to the same handle. In this case, the measurement is dominated by the throughput of loop-back NFSv3 over UDP on the client machine.

6.3 Software distribution

To evaluate how well the read-only file system performs on a larger application benchmark, we compiled (with optimization and debugging disabled) Emacs 20.6 with a local build directory and a remote source directory. The results are shown in Figure 8. The RO experiment performs 1% worse (1 second) than NFSv3 over UDP and 4% better (3 seconds) than NFSv3 over TCP. Disabling integrity checks in the read-only file system (RONV) does not speed up the compile because our caches absorb the cost of hash verification. However, disabling caching does decrease performance (RONC). During a single Emacs compilation, the read-only server consumes less than 1% of its CPU while the read-only client consumes less than 2% of its CPU. This demonstrates that the read-only protocol introduces negligible performance degradation in an application benchmark.

To evaluate how well `sfsrosd` scales, we took a trace of a single client compiling the Emacs 20.6 source tree, repeatedly played the trace to the server from an increasing number of simulated, concurrent clients, and plotted the aggregate throughput delivered by `sfsrosd`. The results are shown in Figure 9. Each sample represents the throughput of playing traces for 100 seconds. Each trace consists of 1,428 RPCs. With 300 simultaneous clients, the server consumes 96% of the CPU.

With more than 300 clients, the FreeBSD server reboots because of a bug in its TCP implementation.

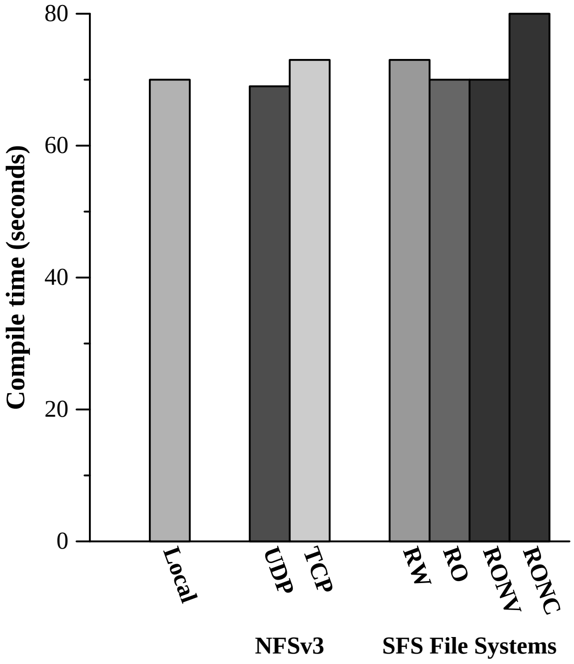

Figure 8: Compiling the Emacs 20.6 source. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warmed server caches, but cold client caches. RW, RO, RONV, and RONC denote respectively the read-write protocol, the read-only protocol, the read-only protocol with no verification, and the read-only protocol with no caching.

We replaced the FreeBSD server with an OpenBSD server and measured that `sfsrosd` maintains a rate of 10 Mbyte/s of file system data up to 600 simultaneous clients.

6.4 Certificate authority

To evaluate whether the read-only file system performs well enough to function as an on-line certificate authority, we compare the number of connections a single read-only file server can sustain with the number of connections to the SFS read-write server, the number of SSL connections to an Apache web server, and the number of HTTP connections to an Apache server.

The SFS servers use 1,024-bit keys. The SFS read-write server performs one Rabin-Williams decryption per connection while the SFS read-only server performs no on-line cryptographic operations. The

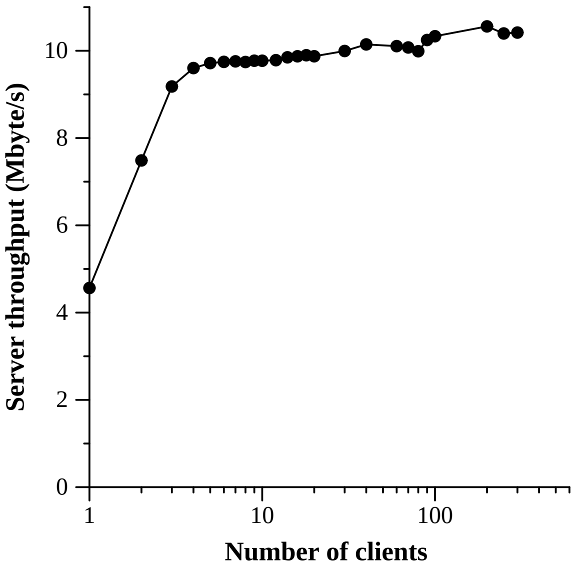

Figure 9: The aggregate throughput delivered by the read-only server for an increasing number of clients simultaneously compiling the Emacs 20.6 source. The number of clients is plotted on a log scale.

Web server was 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 Ephemeral Diffie-Hellman key exchange, DES-CBC3 for confidentiality, and SHA-1 HMAC for integrity.

To generate enough load to saturate the servers, we wrote a simple client program that sets up connections, reads a small file containing a self-certifying path, and terminates the connection as fast as it can. We run this client program simultaneously on two OpenBSD machines. In all experiments, the certificate is in the main memory of the server, so we are limited by software performance, not by disk performance. This scenario is realistic since we envision that important on-line certificate authorities would have large enough memories to avoid frequent disk accesses, like DNS second-level servers.

The SFS read-only protocol performs client-side name resolution, unlike the Web server which performs server-side name resolution. We measured both single-component and multi-component lookups. (For instance, `http://host/a.html` causes a single-component lookup while `http://host/a/b/c/d.html` causes a multi-component

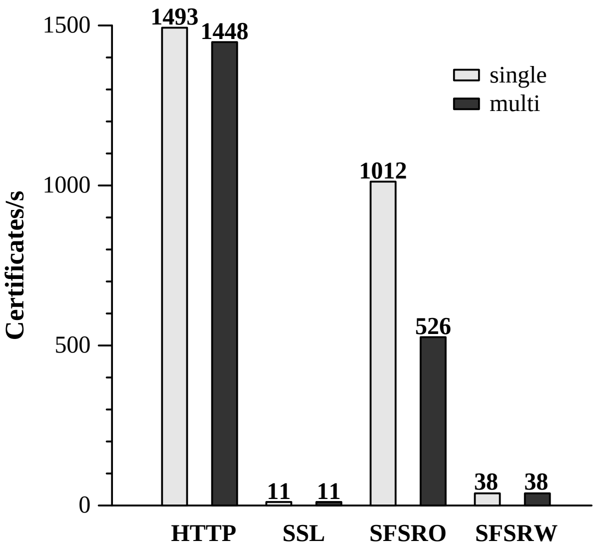

Figure 10: Maximum sustained certificate downloads per second. HTTP is an insecure Web server, SSL is a secure Web server, SFSRW is the secure SFS read-write file system, and SFSRO is the secure read-only file system. Light bars represent single-component lookups while dark bars represent multi-component lookups.

lookup.) The read-only client makes a linear number of RPCs with respect to the number of components in a lookup. On the other hand, the HTTP client makes only one HTTP request regardless of the number of components in the URL path.

The HTTP and SSL single- and multi-component tests consist of a GET `/symlink.txt` and GET `/one/two/three/symlink.txt` respectively, where `symlink.txt` contains the string `/sfs/new-york.lcs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69/`. The SFSRO and SFSRW tests consist of comparable operations. We play a trace of reading a symlink that points to the above self-certifying path. The single-component SFSRO trace consists of 5 RPCs to read a symlink in the top-level directory. The multi-component trace consists of 11 RPCs to read a symlink in a directory three levels deep. The single-component SFSRW trace consists of 6 RPCs while the multi-component trace consists of 12 RPCs.

Figure 10 shows that the read-only server scales well. For single-component lookups, the SFS read-only server can process 26 times more certificate downloads than the SFS read-write server because the read-only server performs no on-line cryptographic operations. The read-write server is bottlenecked by public key decryptions, which each take 24 msec. Hence, the read-write server can at best achieve 38 (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 and slower cryptographic operations on the server than the SFS read-only 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 32% lower than that of the insecure Apache server. In fact, the performance of SFS read-only 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 reasonable to build a distributed on-line certificate authority using SFS read-only servers.

A multi-component lookup is faster with HTTP than with SFSRO. The SFSRO client must make two RPCs per component. Hence, there is a slowdown for deep directories. In practice, the impact on performance will depend on whether clients do multi-component lookups once, and then never look at the same directory again, or rather, amortize the cost of walking the file system over multiple lookups. In any situation in which a single read-only client does multiple lookups in the same directory, the client should have performance similar to the single-component case because it will cache the components along the path.

In the case of our CA benchmark, it is realistic to expect all files to reside in the root directory. Thus, this usage scenario minimizes people's true multi-component needs. On the other hand, if the root directory is huge, then SFS read-only will require a logarithmic number of round-trips for a lookup. However, SFS read-only will still outperform HTTP on a typical file system because Unix typically performs directory lookups in time linear in the number of directory entries; SFS read-only performs a lookup in logarithmic time in the number of directory entries.

7 Conclusion

The SFS read-only file system is a distributed file system that allows a high number of clients to securely access public, read-only data. The data of the file system is stored in a database, which is signed off-line with the private key of the file system. The private key of the file system does not have to be on-line, allowing it to be replicated on many untrusted machines. To allow for frequent updates, the database can be replicated incrementally. The read-only file systems pushes the cost of cryptographic operations from the server to the clients, allowing read-only servers to be simple and to support many clients. An implementation of the design in the context of the SFS global file system confirms that the read-only file system can support a large number of clients, while providing individual clients with acceptable application performance.

8 Acknowledgments

We would like to thank Chuck Blake for his help in setting up the experimental environment, Emmett Witchel for the original SFS read-only implementation, Butler Lampson for shepherding this paper, and Sameer Ajmani, the members of PDOS, and the MIT Applied Security Reading Group for their comments and suggestions.

SFS is free software available from `http://www.fs.net/` or `/sfs/sfs.fs.net:eu4cvv6wcnzscer98yn4qjpjnn9iv6pi/`.

References

[1] 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.

[2] 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.

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

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

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[6] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. Technical report, Massachusetts Institute of Technology, 1999. Submitted for publication, `http://www.pmg.lcs.mit.edu/~castro/application/recovery.pdf`.

[7] D. Duchamp. A toolkit approach to partially disconnected operation. In *Proc. USENIX 1997 Ann. Technical Conf.*, pages 305–318. USENIX, January 1997.

[8] D. Eastlake and C. Kaufman. Domain name system security extensions. RFC 2065, Network Working Group, January 1997.

[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] W. Ford and M.S. Baum. *Secure electronic commerce*. Prentice Hall, 1997.

[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] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, June 1999.

[13] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Advances in Cryptology - Crypto '97*, pages 180–197. Springer-Verlag, 1997. Lecture Notes in Computer Science Volume 1294.

[14] Howard Gobioff, David Nagle, and Garth Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, CMU, June 1999.

[15] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *The eighth Word Wide Web Conference*, Toronto, Canada, May 1999.

[16] U. Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[17] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, 1999. ACM.

[18] R. C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - Crypto '87*, pages 369–378, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science Volume 293.

[19] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[20] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *CCS'99*, Singapore, November 1999.

[21] 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.

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

[23] Sleepycat software. *The Berkeley Database (version 3.0.55)*. `www.sleepycat.com`.

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

[25] TTCP. `ftp://ftp.sgi.com/sgi/src/ttcp/`.

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

[27] C. Wong and S. Lam. Digital signatures for flows and multicasts. In *IEEE ICNP'98*, Austin, TX, October 1998.

[28] www.rpm.org. *RPM software packaging tool*. `www.rpm.org`.