Don't Trust Your File Server

David Mazières and Dennis Shasha
NYU Department of Computer Science
{dm,shasha}@cs.nyu.edu

Abstract

All too often, decisions about whom to trust in computer systems are driven by the needs of system management rather than data security. In particular, data storage is often entrusted to people who have no role in creating or using the data—through outsourcing of data management, hiring of outside consultants to administer servers, or even collocation servers in physically insecure machine rooms to gain better network connectivity.

This paper outlines the design of SUNDR, a network file system designed to run on untrusted servers. SUNDR servers can safely be managed by people who have no permission to read or write data stored in the file system. Thus, people can base their trust decisions on who needs to use data and their administrative decisions on how best to manage the data. Moreover, with SUNDR, attackers will no longer be able to wreak havoc by compromising servers and tampering with data. They will need to compromise clients while legitimate users are logged on. Since clients do not need to accept incoming network connections, they can more easily be firewalled and protected from compromise than servers.

1 Motivation and significance

People are increasingly reliant on outside organizations to manage their data. From data warehouses that store a company's crucial data to consultants performing system administrative duties like backups, more and more data is being left in the hands of people who have no role in creating or using the data. Unfortunately, those who manage data also have the ability to read it or even change it in subtle, difficult to detect ways. Thus, organizations end up placing a great deal of trust in outsiders. Worse yet, the decision to trust outsiders is often driven as much by system management needs as by concern for data security.

At the same time, another trend is developing: people increasingly need to share information with a population of users that cannot be confined by firewalls. Web servers outside of firewalls distribute software and information to the general population. People collocate servers in foreign machine rooms to get better network connectivity. Such outside servers have proven quite vulnerable to attack. While many attacks merely involve defacing web pages to prove a machine has been penetrated, the consequences of a compromise could be much more serious—for instance when a server distributes software many people download and run.

To address these problems, we are building SUNDR—the secure untrusted data repository. SUNDR is a novel distributed data storage system in which the functions of data storage and management can be performed by someone who has no ability to read or modify the data. On client machines, SUNDR behaves like an ordinary network file system, providing reasonable performance and semantics. However, the client cryptographically protects and verifies all data it exchanges with the server. An attacker with complete control of a SUNDR server can accomplish little more than a simple and very noticeable denial of service attack. For any attack to go undetected, the perpetrator would need to compromise a client while an authorized user is logged in.

SUNDR will be the first general-purpose network file system to provide strong data integrity guarantees without trusting the server. Previous work has shown how to protect data secrecy on untrusted servers, or how to protect the integrity of data that is read and written by the same person. In practice, however, many files on a system are readable but not writable by ordinary users. Thus, to replace an existing, general-purpose file system with one that eliminates trust in the server, one must support read-only access to shared data.

SUNDR's design exploits the fact that on modern processors, digital signatures have become cheap enough to compute on every file close. Whenever a client commits a change to the file server, the client effectively digitally signs the entire contents of the file system (though in an efficient, incremental way).

This research was supported in part by National Science Foundation Career award CCR–0093361.

SUNDR will not only protect data integrity, but also make assurances about the recentness of data. Ensuring that users see each others' updates to a file system is a challenging problem when the server is untrusted and the users may not simultaneously be on-line. SUNDR introduces a new notion of data consistency called *relative freshness*: Two SUNDR users will either see all of each others changes to a file system or none. If two users can communicate with each other (for instance, if they are on the same local Ethernet when accessing a remote SUNDR server), they will be guaranteed traditional close-to-open consistency. Otherwise, if the server violates consistency, the users can never again see each others' updates to the file system, thus maximizing the chances of their detecting the attack.

SUNDR's security model makes remote data storage useful where it previously was impractical. However, remote network bandwidth is typically lower than LAN bandwidth. SUNDR will therefore employ large client caches and exploit commonality between files to reduce network traffic. Thus, one can easily "outsource" file service to organizations across the network without heavily penalizing performance, but while also achieving several benefits. The local organization will no longer need to worry about backups (no harm is done if the local cache is lost). Moreover, when several sites share a file system over the wide area network, they will see considerably better performance than with traditional file systems. Finally, because SUNDR does not trust servers, it will allow cooperative caching amongst mutually distrustful clients, and will allow client caches to be used in the reconstruction of server state in the event of a failure.

2 Related work

While a number of file systems have used cryptographic storage in the past, none has provided strong integrity guarantees in the face of a compromised server. The swallow [13] distributed file system used client-side cryptography to enforce access control. Clients encrypted files before writing them to the server. Any client could read any file, but could only decrypt the file given the appropriate key. Unfortunately, one could not grant read-only access to a file. An attacker with read access could, by controlling the network or file server, substitute arbitrary data for any version of a file.

CFS [2] allows users to keep directories of files that get transparently encrypted before being written to disk. CFS does not allow sharing of files between users, nor does it guarantee freshness or integrity of data. It is intended for users to protect their most sensitive files from prying eyes, not as a general-purpose file system. Cepheus [5] adds integrity and file sharing to a CFS-like file system, but trusts the server for the integrity of read-shared data.

The Byzantine fault-tolerant file system, BFS [3], uses replication to ensure the integrity of a network file system. As long as more than $2/3$ of a server's replicas are uncompromised, any data read from the file system will have been written by a legitimate user. SUNDR, in contrast, will not require any replication or place any trust in machines other than a user's client. However, SUNDR and BFS provide different freshness guarantees.

The proposed OceanStore file system [1] names file objects with secure "GUID" handles. For archival objects, which are immutable over all time, the GUID is simply a collision-resistant cryptographic hash of a file's contents, allowing clients to verify the contents directly from the GUID. For mutable objects, the GUID is the hash of a public key and username. Data returned as part of the file must be digitally signed by the private key corresponding to the GUID. File names are mapped to GUIDs using SDSI [14] (effectively using SDSI as a file system).

We actually built a file system (the old read-only file system, described in [10]) in which files are individually signed like mutable OceanStore objects. We subsequently rejected the design for several reasons, however, the most important of which was security. There is no way to guarantee the freshness of files when they are individually signed. If a file was signed a year ago, is that because an attacker wants us to accept a year-old version, or has the user user simply not resigned the file in the past year? To address this problem SUNDR signs file systems rather than individual files, and it introduces the notion of relative freshness.

SUNDR uses hash trees, introduced in [12], to verify a file block's integrity without touching the entire file system. Duchamp [4], BFS[3], SFSRO [6] and TDB [9] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

The CODA file system [7] saves network bandwidth by operating in a disconnected mode, saving changes locally and later reconciling the changes over a fast network link. SUNDR, too, must save bandwidth for people to use servers over the wide area network. However, SUNDR is designed for constant, lower-bandwidth connectivity (e.g., a T1), rather than an intermittent high-bandwidth connection. Like CODA, SUNDR reduces read bandwidth with a large client-site cache. In addition, SUNDR exploits commonality between files to compress data exchanged between clients and servers. The approach is similar to one used by the rsync [16] file transfer utility.

3 Design

SUNDR has several design goals. The server must not see any private file data in the clear. Clients should never be tricked into believing a file contains data that wasn't written there by an authorized user. Furthermore, SUNDR should provide an easy way to recover if a server is indeed compromised. Finally, SUNDR should provide the best guarantees it can on the freshness of data.

Unfortunately, because SUNDR assumes so little about security, there are two attacks it cannot protect against. An attacker can destroy data, wiping or even physically destroying a disk. We call this a *smashing attack*. Second, if users only ever communicate with each other through the file system, an attacker can clone the server, show each user a separate copy, and prevent each user from finding out that the other has updated the file system. We call this a *forking attack*. These vulnerabilities inherently result from letting attackers control a file server. Fortunately, SUNDR can make the attacks easy to detect and recover from.

3.1 Architectural overview

SUNDR will use a two-level architecture reminiscent of Frangipani [15] on Petal [8]. At the lowest level is the SUNDR block server, a network server that stores chunks of data for users. The block server neither understands nor interprets the blocks it is stores for users. This, in large part, is what lets the block server be managed by someone who does not have access to the file system itself. However, the maintainer of the block server can still perform traditional administrative tasks such as making backups and adding more disk space.

The actual file system in SUNDR is built on top of the block server and is implemented entirely by clients. A SUNDR file system is effectively a collection of hash trees—one per user—mapping per-user inode numbers to the contents of files. The nodes of the hash trees are stored at the block server. Given the root directory and the proper decryption keys, a client can, using the block server, fetch and verify the contents of any file in the system.

Every SUNDR file system has a public key, known to all clients accessing the server. SUNDR will use the SFS [11] file naming scheme, which embeds file system public keys in the file namespace. Thus, users can employ any of SFS's key management techniques to obtain a file server's public key. Unlike the current SFS file system, however, a SUNDR server will not know its own private key. The private key will be known only to those with superuser privileges on the file system.

Each user of a SUNDR file system also has a public key. Users' public keys serve two purposes. They authenticate users to the block server, to prevent an unauthorized person from consuming space or deleting blocks still in use. They are also used to digitally sign any changes to the file system, so that other clients can verify the updated contents. The superuser's public key is the one embedded in the file namespace.

Each SUNDR block server also has a public key, called the server key, used for authentication of the server to clients. It may seem odd to authenticate a block server that is supposedly untrusted anyway. However, SUNDR does not guarantee that servers will not misbehave. It only guarantees that bad behavior will be detected, and facilitates recovery. The operator of a block server can then be held accountable, and any damage repaired from backups and client caches. The server key's role is therefore to assure clients they are communicating with a responsible server. Other parties cannot impersonate the server, so users can squarely pin the blame on the operator if anything goes wrong.

3.2 The SUNDR block server

Both for efficiency and because it is not trusted to operate on high-level file constructs, the SUNDR server implements a simple block protocol. The server stores two types of state: user metadata, and file system blocks. The user metadata consists of a table of user public keys and a signed *version structure* for each user, described further in Section 3.3. The superuser's public key is the public key of the file system (the one embedded in the file system's pathname on clients). Note that the term "user" here really designates a user or a group in the traditional UNIX sense. A group is just a SUNDR user with a zero-sized quota (so it cannot allocate space) and a private key known to several people.

The main function of the block server is to store and serve blocks of data that clients can interpret as a file system. The server indexes blocks of data by their cryptographic hashes. A client can store a block at the server and later request the contents of the block by its cryptographic hash. For each block stored, the block server also records a list of users "requiring" the block, and a reference count for each user. When no one requires a block any more, the server garbage collects it and recycles the disk space. Users may have a quota limiting the amount of data they can require a server to store.

Clients and servers communicate over an authenticated link. They negotiate a session key using the server key and the user's public key, and authenticate requests and replies using the session key for a symmetric message authentication code (MAC). Thus, the server knows that a request to store or delete a block really comes from a particular user, and the user knows that an acknowledgment of a store request comes from a server that knows the private half of the appropriate server key.

While the client can send the server an entire block to store, it can also send a description of the block based on existing blocks stored by the server—for instance, to store a new block that is identical to an existing block but for a 20 byte region, or to create a new block by appending the beginning of one block to the end of another. This protocol makes it particularly bandwidth efficient to update all the blocks in a hash tree stored at the server, as a client can simply transmit a new leaf node along with a bunch of small deltas to the parent blocks.

3.3 The SUNDR file system

The SUNDR file system is implemented by clients on top of the block server. The fundamental data structure in the file system is the *virtual inode*, shown in Figure 3.3, with which one can efficiently retrieve and verify any portion of a file. The virtual inode contains a file's metadata and the size and cryptographic hashes of its blocks. If the file is not world-readable, the metadata also includes an index into a table of symmetric encryptions keys to designate the key with which file contents has been encrypted. The table entry can only be decrypted by users with the appropriate private keys. For large files, the inode also contains the hash of an *indirect block*, which in turn contains hashes and sizes of file blocks. The inode further contains the size of all data pointed to by the indirect block. For larger files, an inode can point to double-, triple-, or even quadruple-indirect blocks.

For every user there is an ordered list, known as the *i-table*, mapping 64-bit per-user inode numbers to the cryptographic hashes of virtual inodes. The i-table is broken into blocks, converted to a hash tree, and each node stored at the block server. The hash of the tree's root is known as the user's *i-handle*. The data pointed to by a directory inode is a list of $\langle$file name, user, inode number$\rangle$ triples, sorted by file name.

Each user's version structure consists of that user's i-handle a set of $\langle$user, version$\rangle$ pairs, one for each user of the system. The entire version structure is digitally signed using the user's private key. Thus, given a user's signed version structure, a client can obtain the file contents of any inode number by repeatedly requesting blocks by their cryptographic hash. Inode number 2 in the superuser's i-table has special significance. It is the root directory of the file system.

To update the file system, a client first uploads new blocks to the file server (sending only deltas for blocks in which only a hash has changed). It then brings the user's version structure up to date, putting in the new i-handle and updating version numbers, signing the new structure and sending it to the server. Finally, the client decrements the reference count on any blocks no longer needed.

3.4 Consistency protocol

The goal of SUNDR's consistency protocol is to make it as easy as possible to detect if the server has not faithfully provided a consistent view of the file system to all clients. The SUNDR server is responsible for assigning an order to all file open and close operations (excluding closes of unmodified files). The order assigned by the server must preserve the order of each individual client's operations.

Each user of a SUNDR file system has a version number, incremented each time the user updates the file system. The basic principle behind the SUNDR consistency protocol is that each user signs not only his own i-handle and version number, but also what he believes to be the version numbers of all other users. Thus, each user's version structure contains a version number for every other user as well.

When a user u opens a file, the client increments u's own version number v and signs an *update certificate* with the user's private key, $\{\text{UPDATE}, v + 1\}_{K_u^{-1}}$. When a client closes one or more modified files (more in the case of directory operations such as rename), the update certificate contains the inode numbers of the modified files, $\{\text{UPDATE}, v + 1, I_1, \ldots\}_{K_u^{-1}}$. The client sends the update certificate to the server. The server assigns the operation a place in the order of operations, and sends back any new signed version structures the client has not yet seen. It also sends back other users' update certificates if they are more recent that those users' version structures.

Definition 1 *If x and y are two version structures, we say that $x \geq y$ iff for all users u, $\langle u, v_x\rangle \in x$ and $\langle u, v_y\rangle \in y$ implies $v_x \geq v_y$.*

The client uses the signed messages it receives to update its own version structure s. For each version structure s' that the client sees, it verifies that either $s \geq s'$ or $s' \geq s$. If not, the client declares that a forking attack has occurred. If no attack is detected, the client signs the updated version structure and uploads it to the server.

When a client is determining each user's latest version number, it considers both signed version structures and update certificates forwarded by the server. The client accepts an update certificate so long as the file declared modified in the certificate is not the one being opened. Otherwise, it must wait for the version structure (thereby also obtaining the file owner's new i-handle, and ultimately the new file contents).

Definition 2 *User u_1 has* ***close-to-open consistency*** *with respect to u_2 iff:*

1. *There exists a partial order,* happens before*, on all open and close operations such that any two operations by the same user are ordered, and any close of a file is ordered with respect to all opens and other closes of that same file.*

2. *When u_1 opens file F, and the last close of F to happen before the open was performed by u_2, u_1 will see the contents written by u_2.*

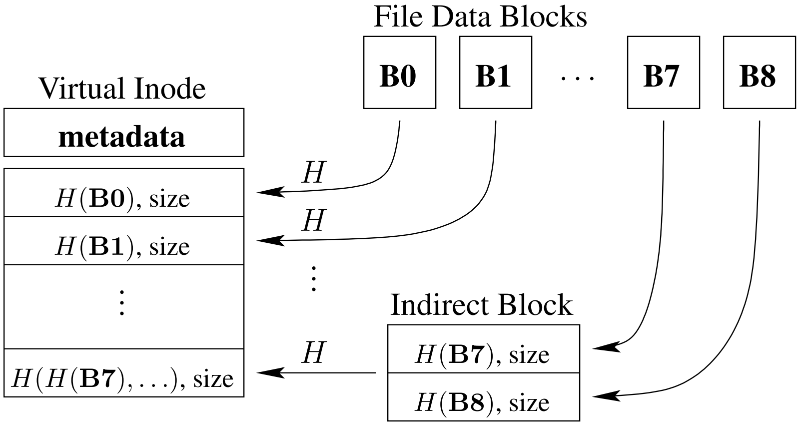

Figure 1. The SUNDR virtual inode structure

Application	savings
dvips	84%
g++	59%
ar	97%

Figure 2. Redundancy in files written by various programs after small changes in the source.

Definition 3 *A file system provides* ***relative freshness*** *iff, whenever user* u_1 *sees the effects of an open or close operation* $\mathcal{O}$ *by* u_2*, then at least until* u_2 *performed* $\mathcal{O}$*,* u_1 *had close-to-open consistency with respect to* u_2*.*

The SUNDR consistency protocol guarantees relative freshness (proof omitted). Given this property, when the clients of two users can communicate on-line, they can achieve close-to-open consistency with respect to each other. The clients need only keep each other up to date about their users' latest version numbers. When each of two users does not know if the other will be on-line, a corrupt SUNDR server can mount a forking attack. However, once such an attack has occurred, the server can never again show either user another update by the other. The corrupt server can therefore partition the set of users and give each partition its own clone of the file system, but when partitioned users have any out-of-band communication—for instance one user in person telling the other to look at a new file—the forking attack will be detected.

3.5 Low-bandwidth protocol

The SUNDR block protocol lets clients use delta-compression to avoid sending an entire metadata block to the server when the new block is identical to an old one but for a small hash value. The same protocol can be used to achieve dramatic compression of file blocks sent to the server under many realistic workloads. The most obvious workload is that of a text editor. When one edits and saves a large file, the editor generally creates and writes a new file substantially similar to the old one.

Many other workloads generate substantially redundant file system traffic, however. To give a sense for this, Figure 2 lists the savings in bandwidth one could obtain from using delta-compression on three typical workloads. The dvips workload consisted of generating a postscript file for this paper twice, with some minor edits in between. 84% (by space) of the write calls made by dvips the second time around were for 8KB blocks that already existed in the first output file (albeit at different offsets). The g++ workload consisted of recompiling and linking a 1.3MB C++ program after adding a debugging print statement. 59% of the writes could have saved using delta compression between the two versions. ar consisted of regenerating a 3.9MB C++ library after modifying one of the source files. 97% of the writes could have been saved.

SUNDR can use the rsync algorithm [16] to reduce bandwidth when one file is very similar to another. However, that still leaves the problem of actually pairing up old files with new ones. In some cases this is easy. For example, dvips just truncates the old file and writes out the new one. Since allocating and freeing of blocks in SUNDR is controlled by the client and decoupled from the file system's structure, the client can, space permitting, temporarily delay removing truncated or deleted files from the server and its local cache in order to use them for delta compression. In other cases, however, the relation between old and new files may be less obvious. The emacs text editor, for instance, saves a file by creating and writing out new file with a different name, forcing that file's contents to be committed to disk, and only then renaming the new file to replace the old.

Fortunately, the SUNDR client can index its local file cache. Using Rabin fingerprints, the client can efficiently compute a running checksum of every (overlapping) 8K block in a file. The client can then index cached files by their n lowest valued checksums for some small n. Many of these checksums will likely appear in a new version of the file to be written out to the server. Thus, the client can pick a file against which to run the rsync algorithm by looking up a small number of fingerprints in its index.

The encryption of blocks slightly complicates delta compression. SUNDR will encrypt data with a block cipher using the combination OFB/ECB mode of [2], but each block stored at the server will have a randomly chosen IV prefixed to the block. When updating internal nodes of the file system, this encryption scheme allows a 20 byte hash in an encrypted block to be updated by changing only 32 bytes of ciphertext.

The situation is more complicated on leaf nodes, when the client is attempting to apply differences between two files in its local cache to an encrypted version of the file on the server. It is for this reason that the SUNDR virtual inode structure permits variable size data blocks and contains the sizes of those blocks. The client can preserve intact any blocks from the old file that are present in the new file, and glue those blocks together with odd-sized fragments. Performing write compression in this way will automatically give other clients the same factor in read compression, as it will maximize the number of the new file's blocks that already exist in their local caches.

4 Summary

The SUNDR file system securely stores data on untrusted servers. Thus, people can base trust decisions on who needs to use data and administrative decisions on how best to manage the data. While some attacks simply cannot be prevented—for instance physical destruction of the hard disk—SUNDR makes it easy to detect and recover from such problems. For instance, after restoring a server from backup, recent changes can securely be merged in from the caches of untrusted clients. SUNDR also introduces the notion of relative freshness—the guarantee that users will see all of each other's changes or none. While weaker than traditional file system consistency guarantees, relative freshness easily lets clients verify tighter consistency guarantees through client-to-client communication.

References

[1] D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, W. Weimer, C. Wells, B. Zhao, and J. Kubiatowicz. Oceanstore: An exteremely wide-area storage system. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000. (to appear).

[2] M. Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[4] D. Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318. USENIX, Jan. 1997.

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

[6] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[8] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92. ACM, October 1996.

[9] U. Maheshwari and R. Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000. (to appear).

[10] D. Mazières. Security and decentralized control in the SFS distributed file system. Master's thesis, Massachusetts Institute of Technology, August 1997.

[11] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999. ACM.

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

[13] D. Reed and L. Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.

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

[15] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, Saint-Malo, France, 1997. ACM.

[16] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1997.