Decentralized User Authentication in a Global File System

Michael Kaminsky, George Savvides, David Mazières, M. Frans Kaashoek
MIT Computer Science and Artificial Intelligence Laboratory
McGill University School of Computer Science, NYU Department of Computer Science
kaminsky@csail.mit.edu, gsavvi1@cs.mcgill.ca, dm@cs.nyu.edu, kaashoek@csail.mit.edu

ABSTRACT

The challenge for user authentication in a global file system is allowing people to grant access to specific users and groups in remote administrative domains, without assuming any kind of pre-existing administrative relationship. The traditional approach to user authentication across administrative domains is for users to prove their identities through a chain of certificates. Certificates allow for general forms of delegation, but they often require more infrastructure than is necessary to support a network file system.

This paper introduces an approach without certificates. Local authentication servers pre-fetch and cache remote user and group definitions from remote authentication servers. During a file access, an authentication server can establish identities for users based just on local information. This approach is particularly well-suited to file systems, and it provides a simple and intuitive interface that is similar to those found in local access control mechanisms. An implementation of the authentication server and a file server supporting access control lists demonstrate the viability of this design in the context of the Self-certifying File System (SFS). Experiments demonstrate that the authentication server can scale to groups with tens of thousands of members.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*authentication, access controls*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*

General Terms

Security, Design

Keywords

Authentication, authorization, groups, users, SFS, file system, ACL, credentials

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010...$5.00

1. INTRODUCTION

A challenge in the design of a global file system, which spans multiple administrative domains, is to provide access control lists (ACLs) that can contain *remote principals*, principals that are defined and maintained by other administrative domains. For example, a user Alice at Microsoft may want to put a *remote user* Bob at Intel on her ACLs for files that are part of a joint development project. As another example, Charles at CMU may want to provide all the students at MIT access to his course software without having to enumerate all of the students on his ACL; he would like to name a *remote group* managed by someone at MIT.

Early network file systems, such as AFS [18], have no mechanism to name remote principals on ACLs; users can place only principals that are in the local cell's authentication database on their ACLs. AFS supports cross-realm authentication through Kerberos [31], but setting up cross-realm authentication requires assistance from system administrators who need to "link" the realms ahead of time. More recently, this centralized trust model has found widespread use in the Windows domain system [24].

Other systems, such as SDSI/SPKI [12, 28] and CRISIS [2], support remote principals through certificates. In these systems, Alice puts Bob's name on her access control list (or a group that contains Bob's name), and Bob presents a set of certificates to prove that he is Bob (or a member of the group). Alice's server then verifies the signatures on the certificates and, if the certificates check out, makes the authorization decision. These systems are powerful (e.g., they provide sophisticated delegation) but require a certificate authentication infrastructure separate from the file system, and, as described in the related work, the process for generating the right chain of certificates that prove access can be complex.

This paper presents an authentication server that supports remote users and groups without requiring certificates. The authentication server provides a simple and familiar interface that mirrors local access control mechanisms. Users create file sharing groups, add local and remote principals to those groups, and then place those groups on ACLs in the file system. When a user accesses this file system, the file server sends a request to its local authentication server to authenticate the user. The authentication server establishes a set of identities for the user and returns them to the file server. We call this set of identities the user's *credentials*. The file server uses these credentials, along with the file's ACL, to make the authorization decision. In this model, users are not responsible for collecting and presenting their credentials to the file system; the local authentication server provides any relevant credentials that the file system might need to authorize the file access.

A design goal for the authentication server is that it can respond to an authentication request without contacting any remote authentication servers (which store the definitions of remote principals).

This goal is important because we do not want file access to be delayed by having to contact a potentially large number of remote authentication servers, some of which might be temporarily unreachable. To achieve this goal, the authentication server pre-fetches and caches remote user and group definitions. This strategy compromises the consistency of group lists; for example, when a remote group changes, this change is not immediately visible at other authentication servers which reference this group. The system has eventual consistency [6], but this consistency may take up to an hour to achieve (in the absence of network failures). We do not believe that this compromise is an issue in practice. Many certificate-based systems make similar trade-offs; for example, certificates might still be usable for hours after revocation [36]. Other systems employ authentication caches that have timeouts [22, 2], or they use security tokens that specify group membership for as long as a user is logged in to his workstation [24].

We have implemented the authentication server for the Self-certifying File System (SFS) [23]. It is fully functional, in daily use, and supports remote users and groups. To validate the functions of the authentication server, we have extended the SFS read-write file system server to support ACLs. For ease of implementation, we store the ACLs for files in the first 512 bytes. This implementation has a performance overhead, but it allows us to demonstrate the usefulness of the authentication server.

The following example demonstrates how Charles at CMU might use this system to share his course software. The details of this example will become clear throughout the rest of the paper, but they provide a flavor of the features provided by our system.

The software is on Charles's ACL-enabled file server in a directory called `/courseware`. First, Charles creates a personal group on the authentication server:

```
$ sfskey group -C charles.cwpeople
```

He can now add members to this new group (the syntax of group members is described in detail in Section 3):

```
$ sfskey group \
   -m +u=james \
   -m +u=liz@bu.edu,gnze6... \
   -m +g=students@mit.edu,w7abn9p... \
   -m +p=anb726muxau6phtk3zu3nq4n463mwn9a \
  charles.cwpeople
```

`james` is a local user, `liz` is a remote user maintained at `bu.edu`, `students` is a remote group at `mit.edu`, and `anb726muxau6phtk3zu3nq4n463mwn9a` is a hash of the public key belonging to a user who is not associated with an organization that runs an authentication server. Both `bu.edu` and `mit.edu` are *self-certifying hostnames* [23]—a combination of the server's DNS name and a hash of its public key. Self-certifying hostnames allow the local authentication server to securely connect to the remote one.

If Charles decides that he wants to share administrative responsibility for his group with a friend at Sun, he can make his friend an owner:

```
$ sfskey group \
   -o +u=george@sun.com,heq38... \
  charles.cwpeople
```

George is now free to add new members (or delete current ones) from Charles's group. Finally, Charles is ready to use his group to share his courseware. He constructs an ACL and places it on the directory as follows:

```
$ cat myacl.txt
ACLBEGIN
user:charles:rwlida:
group:charles.cwpeople:rl:
ACLEND
$ sfsacl -s myacl.txt /courseware
```

Charles has full access permissions to the directory, but the members of his group can only read and list its contents. The details of SFS ACLs are given in Section 4.

The remainder of the paper describes the design and implementation of the authentication server and the ACL-enabled file system. We provide an evaluation of the system and describe the related work. The paper concludes with a discussion of some future work.

2. SECURITY AND TRUST MODEL

SFS is a collection of clients and servers that provide several services—a global file system, remote execution [20], and user authentication. SFS clients and servers communicate using Remote Procedure Calls (RPCs).

SFS depends on public-key cryptography. Servers have private keys, which they do not disclose. Given the corresponding public key, a client can establish a connection to the server that provides confidentiality, integrity and authenticity. In SFS, clients always explicitly name the server's public key using *self-certifying hostnames*, a combination of the server's DNS name and a cryptographic hash of its public key. This paper does not prescribe any specific means for distributing server public keys to clients. A variety of out-of-band methods are possible.

SFS guarantees the following security properties for connections between SFS clients and servers:

- **Confidentiality:** A passive attacker, who is able to observe network traffic, can only accomplish traffic analysis.
- **Integrity:** An active attacker, who is able to insert, delete, modify, delay and/or replay packets in transit, can, at best, only effect a denial of service.
- **Server Authenticity:** When the client initiates a secure connection to the server, the server must prove that it knows the private key corresponding to the public key named in its self-certifying hostname. Once the connection has been established, the client trusts the server to be who it claims to be.

Mazières et al. [23] describes the details of self-certifying hostnames and the protocols that SFS uses to set up secure connections. The next section describes user authentication for requests sent over an SFS connection.

3. USER AUTHENTICATION

SFS allows clients to initiate user authentication any time after the connection is set up. Typically, user authentication occurs right away. Some connections, however, do not require user authentication immediately, and others do not require user authentication at all. User authentication only needs to happen once per connection for a given user (not on every request).

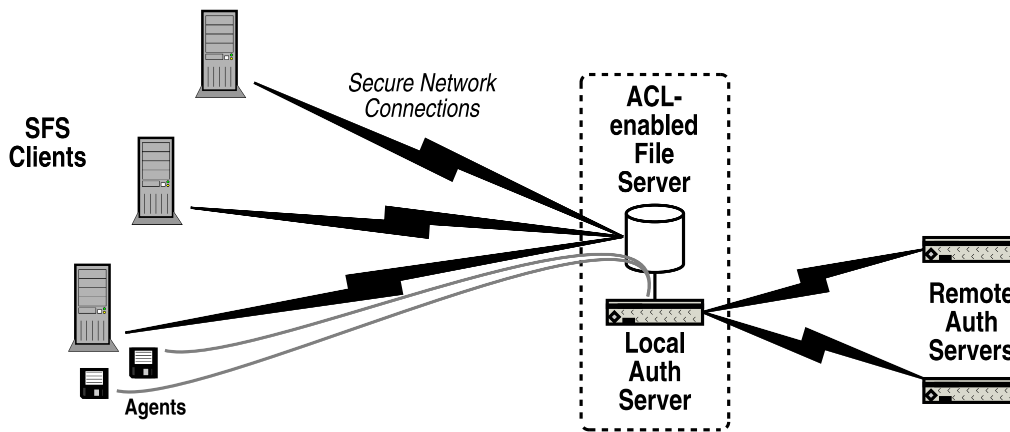

Figure 1: Overview of the SFS authentication architecture

User authentication is a multi-step operation. It begins when the SFS agent, a process running on the user's machine, signs an *authentication request* on behalf of the user with his private key (see Figure 1). The user sends this request to the file server (for example), which passes it, as opaque data, on to the local authentication server. The authentication server verifies the signature on the request and issues credentials to the user based on the contents of its database. The authentication server then hands these credentials back to the file server, which is free to interpret them as it sees fit. Subsequent communication by the user over the same connection receives the same credentials but does not require interaction with the authentication server.

In the current implementation, the local authentication server runs on the same machine as the file server. This detail, however, is not fundamental to the design of the system. Several file servers could share a single "local" authentication server (e.g., on a LAN) by naming it with a self-certifying hostname. Currently, file servers can share an authentication database through replication, restricting updates to a single primary server.

3.1 Authentication server

The SFS authentication server serves two main functions. First, it provides a generic user authentication service to other SFS servers. Second, it provides an interface for users to manage the authentication name space.

Users name remote authentication servers, like other SFS servers, using self-certifying hostnames. They name remote users and groups that are defined on authentication servers in other administrative realms using the same idea. Because remote user and group names contain self-certifying hostnames, they are sufficient to establish a connection to the appropriate authentication server and retrieve the user or group's definition. By naming a remote user or group, users explicitly trust the remote authentication server to define that user or group.

The main challenge in designing the SFS authentication server is how to retrieve remote user, and particularly, remote group definitions. For example, remote authentication servers might be unavailable due to a network partition. Remotely defined groups might themselves include other groups. The nesting could be several levels deep, involving many authentication servers, and possibly cycles.

3.1.1 Interface

The authentication server maintains a database of users and groups in SFS. To a first approximation, this database is analogous to Unix's `/etc/passwd` and `/etc/group`. The authentication server presents an RPC interface which supports three basic operations:

- **LOGIN** allows an SFS server to obtain credentials for a user given an authentication request. **LOGIN** is the main step of the user authentication process described above.
- **QUERY** allows a user (or another authentication server) to query the authentication database for a particular record (e.g., user or group) based on some record-specific key (e.g., name or public key).
- **UPDATE** allows a user to modify records in the authentication database. Access control is based on the record type and the user requesting the update.

LOGIN, by definition, does not require a user-authenticated connection. **QUERY** can be authenticated or unauthenticated; when replying to an unauthenticated **QUERY**, the authentication server can hide portions of the user or group record (see Section 3.2.3). **UPDATE**, however, does require an authenticated connection, so the authentication server can perform access control to the database.

If a user wants to modify a record using **UPDATE**, he first connects directly to the authentication server as he would connect to any other SFS server. The authentication server generates credentials for the user by effectively calling **LOGIN** on itself. Finally, the user issues **UPDATE** over the user-authenticated connection.

3.1.2 User records

Each user record in the authentication database represents an SFS user. Often, SFS user records correspond to Unix user entries in `/etc/passwd`, and system administrators can configure SFS to allow users to register themselves with the authentication server initially using their Unix passwords. SFS user records contain the following fields: [1]

[1] When referring to field names, the convention throughout this paper will be to use a sans-serif typeface.

```
p=bkfce6jdbmdbzfbct36qgvmpfwzs8exu
u=alice
u=bob@cs.cmu.edu,fr2eisz3fifttrtvawhnygzk5k5jidiv
g=alice.friends
g=faculty@stanford.edu,7yxnw38ths99hfpqnibfbdv3wqxqj8ap
```

Figure 2: Example user and group names in SFS. Both `bob` and `faculty` are remote names, which include self-certifying hostnames.

User Record:

User Name	Public Key
ID	Privileges
GID	SRP Information
Version	Audit String

User Name, ID and GID are analogous to their Unix counterparts. Version is a monotonically increasing number indicating how many updates have been made to the user record. Privileges is a text field describing any additional privileges the user has (e.g., "admin" or "groupquota"). SRP Information is an optional field for users who want to use the Secure Remote Password protocol [35]. The Audit String is a text field indicating who last updated this user record and when. Users can update their Public Keys and SRP Information stored on the authentication server with the **UPDATE** RPC.

3.1.3 Group records

Each group record in the authentication database represents an SFS group. Administrators can optionally configure the authentication server to treat Unix groups (in `/etc/group`) as read-only SFS groups. SFS groups contain the following fields:

Group Record:

Group Name	Owners
ID	Members
Version	Audit String

Groups have a name Group Name. SFS groups created by regular users have names that are prefixed by the name of the user. For example, the user `alice` can create groups such as `alice.friends` and `alice.colleagues`. Users with administrative privileges can create groups without this naming restriction. Each group also has a unique ID.

SFS groups have a list of Members and a list of Owners; the group's Owners are allowed to make changes to the group. The elements of these lists are SFS user and group names which are described below. Users who create personal groups implicitly own those groups (e.g., `alice` is always considered an owner of `alice.friends`).

Administrators can set up per-user quotas that limit the number of groups a particular user can create, or they can disable group creation and updates completely. Per-user quotas are stored in the Privileges field of the user record.

3.1.4 Naming users and groups

The authentication server understands the following types of names, which can appear on the Owners and Members lists in group records:

- Public key hashes
- User names
- Group names

Public key hashes are ASCII-armored SHA-1 hashes [13] of users' public keys. They are the most basic and direct way to name an SFS user.

User names refer to SFS user records defined either in the local authentication database or on a remote authentication server. Local user names are simply the User Name field of the record. Remote user names consist of the User Name field plus the self-certifying hostname of the authentication server that maintains the user record.

Similarly, group names refer to group records defined either in the local authentication database or on a remote authentication server. Local group names are the Group Name field of the record, and remote group names are the Group Name field plus the self-certifying hostname of the remote authentication server.

To distinguish between public keys, users, and groups, Owners and Members lists use the following two-character prefixes for each element: `u=`, `g=`, and `p=`. The table in Figure 2 shows several examples of these three types of names. (In the last example, `g=faculty@...` is not prefixed by a user name because it was created by a user with administrative privileges.)

Public key hashes are important for two reasons. First, they provide a universal way to name users who are not associated with an authentication server (e.g., a cable modem user). Second, they can provide a degree of privacy by obfuscating the user names on a group membership list. Such lists of user names might be private; if the user names correspond to email addresses, they might be abused to send spam.

User names are also important because they provide a level of indirection. Naming an authentication server (and *its* public key) instead of naming the user's public key provides a single point of update should the user want to change his key or need to revoke it. Authentication server self-certifying hostnames might appear in more than one membership list, but they typically change less frequently than user keys.

With respect to group names, indirection through an authentication server can provide a simple form of delegation. The last example in Figure 2 shows how a user might name all of the faculty at Stanford. The membership list for that group can be maintained by administrators at Stanford, and SFS users who reference that group do not need to be concerned with keeping it up-to-date. Because all five types of names listed above can also appear on Owners lists, groups with shared ownership are possible. For example, a group might contain the members of a conference program committee. The group's owners are its two co-chairs. The owners and the members of this group all belong to different administrative organizations, but the SFS authentication server provides a unified way to name each of them.

Naming users and groups with self-certifying hostnames delegates trust to the remote authentication server. Delegation is important because it allows the remote group's owners to maintain the

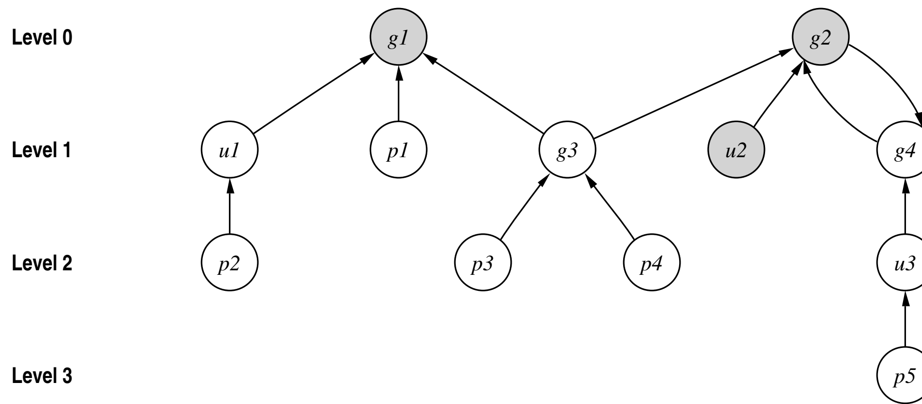

Figure 3: Membership graph for local groups *g1* and *g2*

group's membership list, but it implies that the local server must trust those owners.

3.2 Resolving group membership

The credentials that the authentication server issues may include a list of groups, but these groups must be *local* (defined on the authentication server itself). Any remote groups (or users) of interest to this authentication server must exist as members of some local group. The server *resolves* each local group into a set of public keys by fetching all of the remote users and groups that the local group contains.

Even though this decision puts some restrictions on the user (e.g., by disallowing remote principals to appear directly on ACLs), it has two important advantages. First, the authentication server knows exactly which remote users and groups to fetch (i.e., those which are members of local groups). Second, the authentication server fetches only users and groups that are necessary to issue credentials to a user.

Conceptually, the local groups on a particular authentication server are part of a *membership graph*, which defines the relationship between those local groups, their members, their members' members, and so forth. Each node represents one of the three types of names that can appear in a group's Members list: a public key hash, a user name, or a group name. The direction of the edges indicates membership. For example, an edge from user *U* to group *G* means that *U* is a member of *G*. An edge from public key hash *P* to remote user *U* means that *U* is the user with public key hash *P* (i.e., public key hashes are "members" of a remote user). A multi-hop path in the membership graph means that the membership relationship involves one or more levels of indirection (i.e., it involves remote users or groups that are defined on other authentication servers).

Figure 3 shows the membership graph for two local groups *g1* and *g2*. Local users and groups are shaded gray, remote ones are not. User *u1*, public key hash *p1*, and group *g3* are all members of group *g1*. Group *g3*, user *u2* and group *g4* are members of group *g2*. User *u1* is a remote user with public key hash *p2*; the edge from *p2* to *u1* indicates this relationship (the same relationship exists between *u3* and *p5*). User *u2*, however, is a local user (it resides on the same authentication server as *g1* and *g2*), so it has no in edge. Groups *g4* and *g2* are members of each other and form a cycle in the membership graph.

To issue credentials for a user, the authentication server must determine the local groups to which that user belongs. The server generates this group list by traversing the membership graph starting at the nodes which represent that user: a public key hash and possibly also a local user name (if the user has an account on the authentication server). The authentication server avoids cycles by never visiting the same node twice. When it reaches a local group, it adds that group to the user's credentials. The user with public key hash *p5* would receive credentials containing only the local group *g2*. His credentials would not include group *g1* because the membership graph does not contain a path from *p5* to *g1*.

To construct the membership graph, the authentication server first constructs a complementary graph called the *containment graph*. The containment graph consists of the same nodes, but its edges have the opposite direction. In the containment graph, an edge from group *G* to user *U* means that *G* contains, or lists, *U* as one of its members; similarly, a remote user node "contains" the node for that user's public key hash. Nodes representing a public key hash or a local user do not have any out edges. Nodes representing a remote user have exactly one out edge to the public key hash node for that user. Nodes representing a group (local or remote) have an out edge for each name appearing in the group's Members list.

The containment graph has one or more *levels*. Level 0 consists only of local groups. Level 1 consists of those names that appear directly in the definitions of the local groups. In general, Level *n* consists of all entities that are *n* hops away from the local groups at Level 0. The containment graph can have cycles when a group contains itself through zero or more other groups.

Given the containment graph, constructing the membership graph and traversing it in order to issue credentials is straightforward and efficient. Resolving group membership thus reduces to the problem of constructing the containment graph given a set of local groups. Accurately constructing the containment graph is challenging for the following reasons:

1. *Groups can name remote users and groups.* Constructing the containment graph would be easy if all of the user and group names were local. When membership lists contain remote user and group names, constructing the graph might require contacting a large number of remote authentication servers. Because the graph has dependencies (i.e., fetching a group at Level *n* requires first fetching that group's parent at Level

$n-1$), connecting to authentication servers cannot be done completely in parallel.

2. *Traversing the containment graph must be efficient.* The authentication server cannot simply block while it recursively fetches group lists from other authentication servers. With large groups, which reference many authentication servers, the chance that one or more servers is unavailable could be high. Timing out while resolving group membership is an unacceptable delay when a user is trying to access a file system.

3. *The containment graph changes.* Group membership can be dynamic, particularly since ordinary users can add and remove groups themselves; furthermore, changes can happen to remote groups that are several levels away in the containment graph. The local authentication server needs to know about these changes when it constructs the graph in order to issue credentials accurately.

We address these problems by splitting the authentication task into two parts: constructing the graphs and issuing credentials. The authentication server does the first part, constructing the containment and membership graphs, in the background by periodically pre-fetching and caching the records associated with remote user and group names. It does the second part, issuing credentials, when the user accesses the file system. Because the server has the membership graph cached, it can quickly generate the group membership list for the given user.

Pre-fetching and caching membership information in the background compromises freshness for efficiency, but it allows the authentication server to generate credentials *using only local information*. The local server does not contact other authentication servers in the critical path, when users access the file system. The server saves the cache to disk so that it persists across server restarts.

3.2.1 Updating the cache

To update its cache of user and group records, the local authentication server fetches each remote user and group found in the containment graph. It performs a breadth-first traversal beginning with the local groups and detects graph cycles by remembering which groups it has seen. When the traversal reaches a local user or a public-key hash, the server stops recursing. The authentication server caches all remote user and group lookups (as well as local groups for simplicity). The server also stores the reverse mappings in the cache which represent the membership graph. This cache update cycle is a periodic, background task that currently runs every hour.

Fetching a remote user or group involves securely contacting the remote authentication server that holds the corresponding database record. Because all remote user and group names include a self-certifying hostname, the local authentication server can establish this secure connection without any additional information. The authentication server uses one or more RPCs to download the user record or group membership list.

3.2.2 Cache entries

The cache contains an entry for each node in the containment graph that has an out edge: remote users map to public key hashes and groups map to membership lists. Given the containment graph in Figure 3, the cache would consist of the following entries:

***g1*:** *u1*, *p1*, *g3*
***g2*:** *g3*, *u2*, *g4*
***u1*:** *p2*
***g3*:** *p3*, *p4*
***g4*:** *u3*, *g2*
***u3*:** *p5*

The first two cache entries are for the local groups at Level 0 (*g1* and *g2*). The next three cache entries are the nodes at Level 1 (*u1*, *g3*, and *g4*), and the last entry is for the remote user at Level 2 (*u3*).

The cache also contains reverse mappings that allow the authentication server to traverse the membership graph. The server creates these extra mappings at the time it updates the cache.

3.2.3 Optimizations

The authentication server implements three performance optimizations. First, during each (hourly) update cycle, the local authentication server connects only once to each remote authentication server and stores that connection for later use. Caching connections to authentication servers avoids the public-key operations involved in establishing a secure connection. For a large number of remote users or a large remote group, the savings could be significant. (If SFS used SSL-like session resumption [10, 16], the authentication server would not need to implement this optimization manually.)

Second, authentication servers transfer only the changes that were made to membership lists since the last update. This optimization dramatically reduces the number of bytes a server must fetch to update an existing cache entry. Because the authentication server's cache is written to disk, even when the authentication server first starts, it can benefit from this optimization.

The authentication server implements the incremental transfer optimization using the Version number field present in group records. The local authentication server sends the version number corresponding to the record it has cached, and the remote authentication server responds with the changes that have occurred since that version. (When groups are updated, the server logs the changes by version. If the log is lost or incomplete, the remote server sends back the entire group.)

Third, remote authentication servers have the ability to transform local user names (local to the remote server) into their corresponding public key hashes before returning the membership list. In the example above, when the local authentication server downloads the group *g4* from a remote server *S*, *S* could return *p5* instead of *u3*. *S* knows (or can quickly compute) *p5* because it has the user record for *u3* in its database. This optimization eliminates the need for local authentication servers to fetch the public keys separately for each of those remote users; for groups containing a large number of local users, these additional fetches could be noticeable. As noted in Section 3.1.4, returning public keys also allows administrators to obfuscate a group's membership list. Because group owners want to see user names when they manage their groups, the authentication server does not perform this transformation for authenticated **QUERY** RPCs (**QUERY** RPCs from other authentication servers are not authenticated).

3.2.4 Performance analysis

Two main factors contribute to the performance of updating the cache: the number of bytes that the local authentication server

needs to fetch and the time required to traverse the containment graph. A third factor that can affect performance is the number of public key operations required to update the cache. The performance of updating the cache depends on the resources available both at the client and at the server.

The number of bytes required to download a single remote group's membership list depends on whether or not a copy exists in the cache. If the authentication server does not have a cached copy, the number of bytes is proportional to the size of the group. If it does have a cached copy, the number of bytes is proportional to the number of changes made to the group since the last update cycle. The number of bytes required to fetch all of the remote users and groups in the containment graph depends on the number of remote users and groups.

The time required to update the cache depends on the configuration of the containment graph. Because the authentication server uses a breadth-first traversal, and it downloads each level of the graph in parallel, the latency of the update will be the sum over the maximum download latency at each level.

The number of public key operations (required to set up a new secure connection) is equal to the number of unique servers appearing in the containment graph. Secure connections are cached for an entire update cycle.

3.2.5 Freshness

Aside from using only local information during authentication, the next most important property of the system is freshness. The cache update scheme has eventual consistency—once the update cycle is over, the cache is completely up-to-date (assuming all servers were available).

Given the trade-off between efficiency (using only local information to issue credentials) and freshness, we chose efficiency. Delays during file system access are not acceptable. Given the trade-off between freshness and minimizing the time to update the cache, we chose freshness. An earlier version of the authentication server sought to reduce the amount of time that updates took to complete by fetching records incrementally; we realized, however, that freshness was more important. The time required to complete an update cycle is less critical because updating the cache is a background process.

3.2.6 Revocation

Revocation is closely related to freshness. When a remote user changes his key, or is removed from a remote group record, that change will be reflected in the local cache at the end of the update cycle.

For users who have their public key hashes on another user's group record or ACL (see Section 4.1), revocation is more involved. If such a user wants to revoke his key, he must contact the user who owns the group record or ACL where his key's hash appears, and ask that person to remove it. For this reason, indirecting through a user name is often preferable (if possible).

Revoking the public keys of authentication servers is the most difficult because their self-certifying hostnames might appear in many group records. Section 8 suggest several possible solutions.

3.2.7 Scalability

We designed the authentication server and cache update scheme for a file system context. A typical environment might be a large company or university with tens of thousands of users. Group sizes could range from one or two members up to tens of thousands of members.

For example, MIT's Athena/AFS setup has 19 file servers holding user data, and a total of 20363 file sharing (*pts*) groups.[2] The average number of members per group is 8.77. The number of groups with *n* members, though, declines exponentially as *n* increases. In fact, only 240 Athena groups have more than 100 members. Based on these figures, the SFS authentication server is well-equipped to handle authentication for a large institution, as demonstrated in Section 6.

Though we expect the authentication server to scale to tens of thousands of users and group members, we did not design it to scale to millions. For example, naming all of the citizens of a country or all Visa credit card holders might require a different technique. Such large groups, however, would be unusual for a file system.

The authentication server does, however, need to operate smoothly in the presence of malicious servers that might try to send back an extremely large or infinite group. The local server simply limits the number of users (public key hashes plus local user names) that can appear in the transitive closure of a given local group. In our current implementation, this limit is set to 1,000,000.

3.3 Credentials

Authentication is the process through which the authentication server issues credentials on behalf of users. The user sends an authentication request signed with his private key to the SFS server he is trying to access. The SFS server hands this opaque block of data to the authentication server as part of **LOGIN**, and the authentication server verifies that the user's public key (also part of the authentication request) matches the signature. Given the user's public key, the authentication server uses its database to determine the credentials. The authentication server supports three credential types:

- **Unix** credentials are fields from `/etc/passwd` such as User Name, UID, and GID. The authentication server only issues Unix credentials to users who exists in the authentication database (i.e., registered SFS users who have local accounts). Unix credentials play an important role in the default SFS read-write file system which uses NFS/Unix file system semantics for access control. The SFS remote execution facility uses Unix credentials for the user's shell and home directory as well. The ACL-enabled file system looks only at the User Name field of the Unix credentials (which comes from the User Name field of the user's record in the authentication database).
- **Public Key** credentials are a text string containing an ASCII-armored SHA-1 hash of the user's public key (from his authentication request).
- **Group List** credentials are a list of groups to which the user belongs. Groups in this list correspond to group records in the local authentication database (they do not contain self-certifying hostnames).

The authentication server issues Unix credentials by looking in the database for a user record with the user's public key. If found, the server constructs the Unix credentials from the user record, `/etc/passwd`, and `/etc/group`.

[2] These numbers were accurate as of mid-June 2003.

Permission	Effect on files	Effect on directories
r	read the file	no effect
w	write the file	no effect
l	no effect	change to the directory and list its files
i	no effect	insert new files/dirs into the directory
d	no effect	delete files/dirs from the directory
a	modify the file's ACL	modify the directory's ACL

Figure 4: Access rights available in SFS ACLs

The authentication server issues Public Key credentials by simply hashing the user's public key. Even a user who does not have a user record in the authentication database receives Public Key credentials (provided he has a key pair loaded into his agent and the signature on the authentication request can be verified).

The server issues Group List credentials by checking to see if the user is a member of any local groups in the authentication database. The server traverses the cached membership graph. If a path exists from the user's public key hash or local user name (for users that have Unix credentials) to a local group, the authentication server adds that group to the user's Group List credentials.

4. ACLS AND AUTHORIZATION

Once the user has credentials, the various SFS servers can make access control decisions based on those credentials. For example, the default SFS read-write file system looks only at the Unix credentials and makes requests to the underlying file system as if they were being issued by the local Unix user named in the credentials.

In the SFS ACL-enabled file system, access control is based on all three credential types (Unix, Public Key and Group List). The server checks the ACLs for the relevant files and/or directories to determine if the request should be allowed to proceed.

The SFS ACL-enabled file system, described here and in Section 5.2, is only one example of how a file system could use the extended credentials that the authentication server provides. The essential property that any file system needs is the ability to map symbolic group names to access rights. The authentication server never sees the details of this association; it simply issues credentials in response to the **LOGIN** RPC from the file system. Recently, Linux and FreeBSD have introduced extensions that add ACL support directly in their file systems. These new file systems might provide an alternative to the SFS ACL-enabled file system.

4.1 ACL Entries

An ACL is a list of entries that specify what access rights the file system should grant to a particular user or group of users. SFS ACLs can contain one of four different types of entries. The first three ACL entry types correspond to the credential types that the authentication server can issue to a user.

- **User Names** provide a convenient way to name users with Unix accounts on the local machine. User name ACL entries are matched against the User Name field of Unix credentials.
- **Group Names** refer to SFS groups defined on the local authentication server. Group name ACL entries are matched against each of the groups in the Group List credentials.
- **Public Key Hashes** are ASCII-armored SHA-1 public key hashes which are matched against the Public Key credentials.
- **Anonymous** is an ACL entry type that matches for all users regardless of their credentials.

Remote users and group names cannot appear on ACLs directly. Instead, users can define personal groups (prefixed by their user names) on the authentication server and place remote user and group names on the membership lists of those new groups. This level of indirection provides a single location at which to place the remote authentication server's self-certifying hostname. If the server's public key changes or is revoked, users can update a single group record instead of hundreds or thousands of ACLs spread across the file system. Keeping self-certifying hostnames out of ACLs also helps to keep them small.

4.2 Access rights

We adopted the AFS [18] ACL access rights but extended them to differentiate between files and directories (AFS only has ACLs on directories). Figure 4 lists the different permissions that an ACL can contain and the meaning that each permission has. Unlike AFS, the ACL server does not support negative permissions; once an ACL entry grants a right to the user, another entry cannot revoke that right.

5. IMPLEMENTATION

SFS clients and servers communicate using Sun RPC [30]. When the client and server are located on different machines, the RPCs travel over a transport that provides confidentiality, integrity and authenticity [23]. This secure channel is set up using public-key cryptography (with the public key hash in the self-certifying hostname). On the same machine, SFS uses file descriptor passing to hand off RPC connections.

5.1 Authentication server

The SFS authentication server implements an RPC interface (described in Section 3.1.1) to its authentication database. To improve scalability, the server has a Berkeley DB [3] backend, which allows it to efficiently store and query groups with thousands of users. By using a database, the authentication server can scale to tens of thousands of users per group. The authentication server also uses Berkeley DB to store its cache.

5.2 ACL-enabled file system

The SFS ACL-enabled file system is an extension of the SFS read-write file system. Both variants store files on the server's disk using NFSv3 [8]. This technique offers portability to any operating system that supports NFSv3 and avoids the need to implement a new in-kernel file system.

```
$ sfskey group author1.sosp2003@stanford.edu,7yxnw38ths99hfpqnibfbdv3wqxqj8ap
 Group Name: author1.sosp2003@stanford.edu,7yxnw38ths99hfpqnibfbdv3wqxqj8ap
         ID: 100004
    Version: 1
     Owners: <none>
    Members: u=author1
             u=author2@cs.cmu.edu,fr2eisz3fifttrtvawhnygzk5k5jidiv
             u=author3@berkeley.edu,5uvx79hddyqbrrjk6d5mx8zmgsujvmin
      Audit: Last modified Fri, 21 Mar 2003 13.43.48 -0800 by author1@1.2.3.4
```

Figure 5: Using `sfskey` to view an SFS group.

5.2.1 Locating ACLs

The drawback to storing files on a standard Unix file system (through NFS) is that Unix file systems do not provide a convenient location to store ACLs. Our implementation stores file ACLs in the first 512 bytes of the file and directory ACLs in a special file in the directory called `.SFSACL`.

Our implementation decisions are motivated by the need to have an efficient way to locate the ACL for a file system object given an NFS request for that object. Because NFS requests contain *file handles* and not path names, storing a mapping between path names and ACLs in an external database indexed by path name was not practical. Storing the ACLs in a database indexed by NFS file handle would technically be possible, but it would impose constraints on how one treated the underlying file system. Moving files around, backups, and restores, for example, would all need to preserve disk-specific meta data such as inode number to ensure that the NFS file handles did not change (or tools would need to be written which updated the ACL database appropriately).

To retrieve a file's ACL, the SFS ACL file system server can simply read the first 512-bytes of the file to extract its ACL. The server hides the ACL's existence from the client by doctoring the NFS RPC traffic as it passes through the server (e.g., adjusting the `offset` fields of incoming **READ** and **WRITE** requests and the `size` field of the attributes returned in replies).

To retrieve a directory's ACL, the ACL file system server issues a **LOOKUP** request for the file `.SFSACL` in the given directory. **LOOKUP** returns the NFS file handle for the `.SFSACL` file which contains the directory's ACL in its first 512-bytes. The server proceeds to read the ACL as in the file case above.

5.2.2 ACL Format

For rapid prototyping and to simplify debugging, we chose to use a text-based format for the ACLs. A sample ACL for a directory containing the text of an SOSP conference paper submission might be

```
ACLBEGIN
sys:anonymous:rl:
group:author1.sosp2003:wlida:
ACLEND
```

The user `author1` created a group on his local authentication server and placed that group on the ACL for the directory with all of the files related to the paper. Permissions such as `r` and `w` have no effect on directories, but they are still important because any new files created in this directory will inherit this ACL. Thus, users can determine the default access permissions on files when they create the directory.

The on-disk ACL representation could be more compact. In the future, we may move to a binary format to encode the ACL.

5.2.3 Permissions

When the SFS ACL file system server receives an NFS request, it retrieves the necessary ACL or ACLs and decides whether to permit the request. The access rights required for a given request are based on the type of that request and the object(s) involved. The SFS client can determine these rights (for instance, when opening a file) by issuing the NFSv3 **ACCESS** RPC. The SFS ACL file system replies to **ACCESS** based on an object's ACL.

File attributes still contain the standard Unix permission bits. Though these bits are not used in determining file permissions, the SFS ACL server sets them to the nearest approximation of the file's ACL. When the correct approximation is ambiguous, the server uses more liberal permissions. Client applications might otherwise fail due to a perceived lack of access rights even though the request would actually succeed based on the ACL.

5.2.4 Caching

The ACL-enabled file system server maintains two internal caches to improve performance. First, the server caches ACLs to avoid issuing extra NFS requests to the underlying file system every time it needs to retrieve an ACL. Because most NFS requests from the client will require an ACL to decide access, the ACL cache could reduce the number of extra NFS requests by a factor of two or more.

Second, the server caches the permissions granted to a user for a particular ACL based on his credentials. The permissions cache avoids reprocessing the ACL which might be expensive if the user has many credentials (i.e., he is a member of many groups).

5.3 Usage

We provide several tools for manipulating groups and ACLs. Based on the SOSP submission example given in Section 5.2.2, this section demonstrates how to use the system.

We've extended the `sfskey` tool to view, create, and update group lists on an authentication server. Figure 5 shows the output of the view command.

A new tool, `sfsacl`, allows users to view and set ACLs from their clients. The `sfsacl` program uses two special RPCs which are not part of the standard NFS/SFS protocols; the ACL-enabled file system server intercepts these RPCs and handles them directly

(instead of passing them to the underlying NFS loopback file system). The SFS client software provides a way for `sfsacl` to obtain both a connection to the ACL file server and the file handle of the directory containing the ACL it wants to access. (The user accesses a special file name in the directory, and the client file system software creates a special symbolic link on-the-fly whose contents is the desired file handle).

Aside from `sfskey` and `sfsacl`, all of the implementation is on the server. The standard SFS read-write file system client is completely compatible with the SFS ACL-enabled server.

6. EVALUATION

We provide an evaluation of the scalability of the authentication server and the performance of the ACL-enabled file system.

6.1 Authentication server

The number of bytes that the authentication server must transfer to update its cache depends on the number of remote records that it needs to fetch. The number of bytes in each user record is essentially constant. (Some fields such as the User Name and the Audit String have a variable length, but the difference is small.) For each group record, the number of bytes that the server needs to transfer depends either on the size of the group (for new, uncached groups) or the number of changes (for out-of-date, cached groups). We currently do not compress any network traffic, but compression could produce significant savings.

An authentication server fetches a group record using the **QUERY** RPC. Because our implementation fixes the maximum size of a single RPC message, we limit the number of members plus owners (for uncached groups) or the number of changes (for cached groups) that are sent back in the reply to 250. Larger groups require more than one **QUERY** RPC.

Connecting to the remote authentication server also requires two RPCs to establish the secure channel. Because the implementation caches connections, it only establishes one such channel per update cycle; therefore, subsequent queries to a given authentication server do not send these initial RPCs (or perform the associated public key operations). The numbers given below do not include the bytes transferred due to these RPCs, which total just over 900 bytes.

We ran two experiments to measure the number of bytes transferred when fetching group records. In the first experiment, the local authentication server fetched the entire group because it did not have a version in its cache. Unsurprisingly, the number of bytes transferred scales linearly with group size. The total number of groups transferred in this experiment was 1001. Each group consisted of an increasing number of users: 0, 10, 20,..., 9990, 10000. Users in this experiment were represented by the hashes of their public keys (34 bytes each). The group names were all 16 bytes, and the audit strings were 70 bytes. The owners list was empty.

In the second experiment, the local authentication server had a cached copy of the group. As expected, the number of bytes transferred scales linearly with the number of changes to the group. In this experiment, we varied the number of changes that the server had to fetch from 0 to 9990 by ten. Each change was the addition of a new user (35-bytes: the symbol "+" and a public key hash). Here, the audit strings were slightly shorter (approximately 65 bytes).

The following table shows sample data from these experiments. Q is the size of the RPC request, R is the size of the reply, M is number of users in the group (or the number of of changes to the group), S is the size of a single user (or change) and O is the RPC overhead incurred for each additional 250 users. B is the total number of bytes transferred. All values (except for M) are in bytes.

To transfer	Q	R	S	M	O	B
0 users	72	136	40	0	216	208
10000 users	72	136	40	10000	216	408632
0 changes	72	108	40	0	180	180
1000 changes	72	108	40	1000	180	40720

The experimental results show that the total number of bytes transferred for a particular group size or number of changes is given by the following formula:

$$B = Q + R + (M \times S) + \left\lfloor \frac{M}{251} \right\rfloor \times O$$

The values of Q, R, S, and O are constants that depend on the characteristics of the group, such as the length of member names or the audit string.

These experiments also show that the RPC overhead is insignificant. For example, to transfer 10000 users requires approximately 400 KB. The total RPC overhead is only $\lfloor \frac{M}{251} \rfloor \times O = 8424$ bytes, which is just over 2% of the total bytes transferred.

These numbers demonstrate that the authentication server can reasonably support the group sizes found on MIT Athena. The largest Athena group has 1610 members. Based on the formula above, the number of bytes required to transfer that group is 65904.

6.2 ACL-enabled file system

The ACL mechanism introduces a penalty in the overall performance relative to the original SFS read-write file system. This penalty is mainly due to the extra NFS requests that the ACL file system needs to issue in order to locate and read (or write) the ACLs associated with the incoming requests from the client.

6.2.1 Methodology

We measured file system performance between a server running Linux 2.4.20 and a client running FreeBSD 4.8. The machines were connected by 100 Mbit/s switched Ethernet. The server machine had a 1 GHz Athlon processor, 1.5 GB of memory, and a 10,000 RPM SCSI hard drive. The client machine had a 733 MHz Pentium III processor and 256 MB of memory.

We used the Sprite LFS small file micro benchmark [29] to determine the performance penalty associated with our ACL mechanism. The benchmark creates, reads, and deletes 1,000 1024-byte files. The benchmark flushes the buffer cache, but we set the sizes of the internal ACL and permission caches to large enough values to make sure that entries were not flushed from the caches by the time they were needed again. By ensuring that the caches do not overflow during the test, we can better understand how the performance penalty relates to the extra NFS calls and permissions checks that are needed for access control. Using this test, we measured the performance of the original SFS file system, the ACL-enabled file system, and the ACL-enabled file system with the caches turned off.

6.2.2 Results

Figure 6 shows the results of running the benchmark on the three file system variants. The figure breaks down the best result of five trials.

Phase	Original SFS seconds	ACL SFS with caching seconds (slowdown)	ACL SFS without caching seconds (slowdown)
create	15.9	18.1 (1.14X)	19.3 (1.21X)
read	3.4	3.5 (1.03X)	4.3 (1.26X)
delete	4.8	5.1 (1.06X)	6.0 (1.25X)
Total	24.1	26.7 (1.11X)	29.6 (1.23X)

Figure 6: LFS small file benchmark, with 1,000 files created, read, and deleted. The slowdowns are relative to the performance of the original SFS.

In the create phase, the performance penalty is due mainly to the extra NFS requests needed to write the ACL of each newly created file. Processing the ACL to check permissions also contributes to the performance penalty.

For the read and delete phases of the benchmark, the ACL cache offers a noticeable performance improvement. The server can avoid making NFS calls to retrieve ACLs because they are cached from the create phase of the benchmark.

Figure 7 shows the cost of reading a file during the read phase of the benchmark, expressed as the number of NFS RPCs to the loopback NFS server. Each **READ** request from the client is preceded by a **LOOKUP** and an **ACCESS**. In the current implementation, without caching, **LOOKUP** and **ACCESS** each require two extra NFS RPCs to determine the directory's ACL and the file's ACL. **READ** requires one extra RPC to determine the file's ACL.

With caching enabled, the actual read-phase performance slowdown (1.03X) basically agreed with the predicted slowdown (1.00X). The difference is likely due to the overhead of cache lookups.

With caching disabled, the actual slowdown (1.26X) was much lower than the predicted slowdown (2.67X). We attribute this discrepancy to the fact that the operating system on the server is reading the entire file into its buffer cache when it reads the file's ACL (stored in its first 512-bytes). When the operating system reads the file's contents, it does not need to access the disk.

These experiments indicate that the additional NFS requests required to support ACLs result in a performance slowdown. We chose the small file benchmark in particular to expose the performance overhead of having to retrieve ACLs through NFS loopback.

We expect that many end-to-end applications will experience a minimal performance impact due to the introduction of ACLs. For example, we ran an experiment that involved unpacking, configuring, compiling, and deleting an Emacs distribution on both the original SFS and on the ACL-enabled SFS (with caching). We found that the total slowdown was only 1.02X.

7. RELATED WORK

This section discusses the three main approaches to user authentication taken by previous systems: centralized authorities, certificate-based systems, and systems that expose public keys.

7.1 Centralized authentication

The **Kerberos** authentication system [31] combined with the **AFS** [18] file system provides a secure distributed file system with centralized user authentication. Kerberos principals are organized into realms which are usually defined along administrative boundaries. AFS has no mechanism to refer to remote principals; users can only place principals that are in the local cell's authentication database in their ACLs. Cross-realm authentication through Kerberos allows remote users to appear on local ACLs but requires the remote user to first register himself in the local realm. Registration only works if the two realms have been "joined" ahead of time. Because Kerberos is based on shared-secret cryptography, joining two realms is a complicated operation which requires coordination between the two realm administrators. Kerberos itself has no support for groups of principals. AFS has basic support for local groups which can contain local principals; these groups can appear on ACLs.

Microsoft Windows 2000 servers [24] support a sophisticated distributed computing infrastructure based on collections of *domains*. Windows domains combine to form domain trees and forests (sets of domain trees under a single administrative authority), which have automatic two-way transitive trust relationships between them based on Kerberos V5. Administrators can manually set up explicit, non-transitive, one-way trust relationships between different forests. Unlike SFS, the Windows domain system requires an organized trust infrastructure.

Windows 2000 supports several types of groups, each with different semantics and properties. Some groups can contain members only from the local domain but can be placed on ACLs in other domains. Other groups can contain members from other domains but can be placed only on local ACLs. A third type of group can contain members from any domain and can be placed on any ACL in the domain tree, but these universal groups require global replication. The Windows domain system uses this combination of group types to reduce login time and limit the amount of replication required. SFS provides a single group type, which can contain members defined on any authentication server and which can be placed on any group list (and thus any ACL).

7.2 Certificate-based systems

The **Taos** operating system [22, 34] and the Echo file system [4] provide a secure distributed computing environment with global naming and global file access. Echo supports globally named principals (users) and groups on access control lists.

User authentication in Taos is based on certificate authorities (CAs) which "speak for" named principals, issuing certificates which map a public key (associated with a secure channel) to a name. In a large system where not everyone trusts a single authority, the CAs can be arranged into a tree structure which mirrors the hierarchical name space. Authentication in such a system involves traversing a path in the authority tree from one principal to another through their least common ancestor. This traversal establishes a chain of trust through a series of CAs. Gasser et al. [17] and Birrell et al. [5] discuss similar hierarchic authority trees and suggest "symbolic links" or "cross-links" as a way to allow one CA

NFS request	Original SFS (NFS RPCs)	ACL SFS with caching (NFS RPCs)	ACL SFS without caching (NFS RPCs)
lookup	1	1	3
access	1	1	3
read	1	1	2
Total	3	3	8
Predicted slowdown	1.00X	1.00X	2.67X

Figure 7: Cost of reading a file during the read phase of the Sprite LFS small file benchmark, expressed as the number of NFS RPCs to the loopback NFS server.

to directly certify another without traversing a path through their common ancestor.

Taos certifies public keys using certificates and a simple proof system. The SFS user authentication model does not use certificates or have CAs (in the traditional sense). Instead, SFS exposes public keys to the user in two ways: group records and ACLs can contain public key hashes, and remote user and groups names can contain public keys of authentication servers in the form of self-certifying hostnames. Taos, however, insists that ACLs contain human-sensible names and relies on one or more CA to give meaning to those names.

SPKI/SDSI [12, 28] provides a sophisticated, decentralized public-key naming infrastructure that can be used in other systems, and it offers a number of advantages over strict hierarchical PKIs, such as X.509 [36], and web-of-trust based systems such as PGP [38]. SPKI/SDSI principals are essentially public keys, which define a local name space. Users that want to authenticate themselves to a protected resource must prove they have access to that resource by providing a certificate chain. SPKI/SDSI requires complex proof machinery and algorithms to discover and validate certificate chains [9]. No truly distributed implementations of SDSI exist because distributed chain discovery algorithms are difficult to deploy.

SFS users name principals more directly using public keys and self-certifying hostnames. SPKI/SDSI certificates (using meta data) offer more fine-grained control than SFS can provide, but we argue that in a file system context, public keys and self-certifying hostnames are sufficient. SDSI only uses local information (plus the client-provided certificate chain) to decide group membership; SFS also takes this approach of not contacting other entities when issuing credentials.

CRISIS [2], the security component of WebOS [32], is also a certificate-based system, which itself is based heavily on the Taos authentication architecture. Unlike Taos (but like SDSI), principals who want to authenticate to a server are responsible for presenting to the server all of the necessary certificates. The server verifies the signatures on these certificates by making sure that the public keys are endorsed by trusted CAs. CAs are arranged hierarchically, and the server checks that there exists a path of trust through this CA hierarchy from the local domain to the domain of each of the signing principals. Principals can create new groups by issuing an identity certificate for the group and then signing transfer certificates to all of the group's members.

The **Snowflake** project [19] extends the SPKI [11] implementation by Morcos [26] in several ways to provide end-to-end authorization across administrative boundaries, levels of abstraction, and existing protocols. Snowflake is also based on the Taos model. End points, rather than intermediary nodes, make all the authorization decisions. In this way, the end point receives requests for authorization unadulterated by intermediary nodes such as protocol translators. Our approach with SFS also follows an end-to-end authorization model, but does not require a separate PKI, which in Snowflake adds 60 msec to the 50 msec required to send a request over SSL [10, 16].

The **Grid Security Infrastructure (GSI)** [7, 15], part of the **Globus** [14] project, is designed to provide inter-domain authentication and authorization in a distributed computing environment. GSI is a certificate/CA-based system which uses certificate chains to authenticate entities. A central idea in GSI is that users can have both local and global identities. GSI authenticates the user's global name and then maps it onto a site-specific, local one. The site's local security infrastructure (e.g., Kerberos) can authorize access to resources based on the local name that it knows. GSI provides a way for sites to retain independent control of their resources but still participate in a global computing environment.

The **Web** allows individuals to use personal SSL certificates to authenticate to Web sites. These personal certificates are used rarely in practice, however, because no universally accepted public key infrastructure exists to issue and verify them. Instead, Web site rely on local databases, passwords, and "cookies" to authenticate users.

OceanStore [21] is a wide-area file system that has adapted self-certifying hostnames to name objects but uses the locally linked name spaces from the SDSI framework for secure bindings.

7.3 Exposing Public Keys

The idea of exposing public keys to users has its basis in earlier systems. **PGP** [38] first popularized this idea by providing a standard way to encode and distribute PGP public keys. PGP fingerprints (hashes of the public key) give users a convenient way to verify keys they obtain with the keys' owners. PGP key servers provide a centralized location for and interface to key distribution.

SSH [37] also allows users to directly manipulate public keys in the context of remote login. SSH keeps an ASCII version of the user's public in a file called (historically) `identity.pub` in the user's home directory. By placing the contents of this file into the `authorized_keys` file on a remote machine, the user can use public-key cryptography to authenticate to the server. The `authorized_keys` file acts like an ACL for remote login.

In the context of file systems, **Farsite** [1] uses ACLs to authorize write access for users to directory groups. These ACLs contain the public keys of users who are permitted to write to the directory and its files. Farsite, however, does not cross administrative domains; its target is a large corporation or university.

CapaFS [27] and **DisCFS** [25] use capabilities to provide remote users with access to the local file system. In these systems, like SFS, users do not need an account on the local file server in order to access files. SFS, however, uses ACLs to perform access

control, providing users with a more traditional file sharing interface. File owners share files by creating groups, adding users (or other groups) to those groups, and then placing those groups on ACLs. Clients can access files as usual (i.e., they do not need to present per-file or per-directory capabilities).

LegionFS [33] is a wide-area file system that embeds public keys into object names, similar to self-certifying hostnames. It also provides ACLs for objects, with permissions per object method. A LegionFS client can dynamically modify whether secure communication should be used or not. The paper is unclear, however, if LegionFS can name users and groups securely.

8. FUTURE WORK

Currently, the authentication server updates its cache once per hour. We would like to add the ability to update cache entries more or less frequently by including a refresh value with each user and group record, as in DNS. Local authentication servers can use this value to see if the entry needs updating before fetching it again; consequently, remote administrators can control the freshness of cached public key hashes and membership lists. Records would also include a timeout value that tells local servers how long to keep using a cache entry if they cannot contact the remote authentication server.

Remote user and group names provide a level of indirection so that a user's public key can be stored in a single location. Self-certifying hostnames for authentication servers, however, can appear in multiple group records. If the server's key changes and the old key needs to be revoked, each of those records needs to be updated. The SFS file system uses symbolic links to add a level of indirection so that users do not need to refer to servers by their self-certifying hostnames. The authentication database might also allow users to name remote authentication servers through symbolic links in the file system. When refreshing the cache, the server would traverse the symbolic links to determine the self-certifying hostname of the remote authentication server. To change or revoke the server's key, one needs only to change a single symbolic link.

Simply revoking a key for an authentication server (as opposed to changing it) is potentially an easier problem. First, authentication servers can maintain revocation lists of keys that are no longer valid. Second, during the periodic cache refreshes, the authentication servers could exchange SFS-style key revocation certificates [23]. These certificates are self-authenticating in that they are signed with the private key of the server whose key is being revoked; therefore, the receiving authentication server does not need to worry about the trustworthiness of the server from which the certificate came. Once the authentication server verifies the revocation certificate, it can add that key to its personal revocation list and refuse to contact remote authentication servers with that key.

The authentication server and ACL-enabled file system support *user-centric* trust; individual users can set up trust relationships by placing remote principals into local groups. In many environments (e.g., academia or free-software development communities), this flexibility is welcome. In some environments (e.g., large corporations), administrators might want to have a site policy that restricts users from naming certain remote principals. SFS could provide administrators with this ability by allowing them to install blacklists or whitelists. These lists could have patterns that the authentication server matches against remote principals before fetching them.

9. CONCLUSION

This paper contributes a new design point in the space of user authentication. We sacrifice generality for ease-of-use and simplicity of implementation. Our system combines the ease-of-use found in centralized authentication systems with the unified naming semantics for remote principles common in designs based on certification hierarchy. Our authentication server lacks the generality found in certificate-based systems, but it benefits from a simpler implementation, which does not require an infrastructure for managing certificates. By pre-fetching and caching remote principals, authentication servers can issue credentials without contacting remote sites during file access. Experiments demonstrate that the server can scale to groups with tens of thousands of users. The implementation is part of the open-source SFS distribution, available at `http://www.fs.net/`.

10. ACKNOWLEDGMENTS

We are grateful to our shepherd Ted Wobber for his valuable input and feedback. We also thank Chuck Blake, Butler Lampson and the anonymous reviewers for their comments and suggestions. This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract #N66001-01-1-8927, and MIT Project Oxygen.

REFERENCES

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, Boston, MA, December 2002.

[2] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium*, pages 15–30, San Antonio, TX, January 1998.

[3] Berkeley DB. `http://www.sleepycat.com/`.

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

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

[6] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[7] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

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

[9] Dwaine Clarke. SPKI/SDSI HTTP server/certificate chain discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, September 2001.

[10] T. Dierks and C. Allen. The TLS protocol. RFC 2246, Network Working Group, January 1999.

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylönen. SPKI certificate theory. RFC 2693, Network Working Group, September 1999.

[12] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylönen. SPKI certificate documentation. Work in progress, from `http://www.pobox.com/~cme/html/spki.html`, 2002.

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

[14] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11 (2):115–128, 1997.

[15] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, San Francisco, CA, November 1998.

[16] 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.

[17] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th NIST-NCSC National Computer Security Conference*, pages 305–319, Baltimore, MD, October 1989. URL `citeseer.nj.nec.com/gasser89digital.html`.

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

[19] Jon Howell and David Kotz. End-to-end authorization. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 151–164, San Diego, CA, October 2000.

[20] Michael Kaminsky, Eric Peterson, Kevin Fu, David Mazières, and M. Frans Kaashoek. REX: Secure, modular remote execution through file descriptor passing. Technical Report MIT-LCS-TR-884, MIT Laboratory for Computer Science, January 2003.

[21] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

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

[23] 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.

[24] Microsoft Windows 2000 Advanced Server Documentation. `http://www.microsoft.com/windows2000/en/advanced/help/`.

[25] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Secure and flexible global file sharing. In *Proceedings of the USENIX 2003 Annual Technical Conference, Freenix Track*, pages 165–178, San Antonio, TX, June 2003.

[26] Alexander Morcos. A java implementation of simple distributed security infrastructure. Master's thesis, Massachusetts Institute of Technology, May 1998.

[27] Jude Regan and Christian Jensen. Capability file names: Separating authorisation from user management in an internet file system. In *Proceedings of the 10th USENIX Security Symposium*, pages 221–234, Washington, D.C., 2001.

[28] Ronald L. Rivest and Butler Lampson. SDSI—a simple distributed security infrastructure. Working document from `http://theory.lcs.mit.edu/~cis/sdsi.html`, 2002.

[29] 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.

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

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

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

[33] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the IEEE/ACM Supercomputing Conference (SC2001)*, November 2001.

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

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

[36] X.509. *Recommendation X.509: The Directory Authentication Framework*. ITU-T (formerly CCITT) Information technology Open Systems Interconnection, December 1988.

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

[38] Philip Zimmermann. *PGP: Source Code and Internals*. MIT Press, 1995.