User Authentication and Remote Execution Across Administrative Domains

by

Michael Kaminsky

B.S., University of California, Berkeley, 1998
S.M., Massachusetts Institute of Technology, 2000

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author . . . .
Department of Electrical Engineering and Computer Science
June 30, 2004

Certified by . . . .
M. Frans Kaashoek
Professor, Department of Electrical Engineering and Computer Science, MIT
Thesis Supervisor

Certified by . . . .
David Mazières
Assistant Professor, Department of Computer Science, New York University
Thesis Supervisor

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

User Authentication and Remote Execution
Across Administrative Domains
by
Michael Kaminsky

Submitted to the Department of Electrical Engineering and Computer Science
on June 30, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A challenge in today's Internet is providing easy collaboration *across administrative boundaries*. Using and sharing resources between individuals in different administrative domains should be just as easy and secure as sharing them within a single domain. This thesis presents a new authentication service and a new remote login and execution utility that address this challenge.

The authentication service contributes a new design point in the space of user authentication systems. The system provides the flexibility to create *cross-domain groups* in the context of a global, network file system using a familiar, intuitive interface for sharing files that is similar to local access control mechanisms. The system trades off freshness for availability by pre-fetching and caching remote users and groups defined in other administrative domains, so the file server can make authorization decisions at file-access time using only local information. The system offers limited privacy for group lists and has all-or-nothing delegation to other administrative domains via nested groups. Experiments demonstrate that the authentication server scales to groups with tens of thousands of members.

REX contributes a new architecture for remote execution that offers extensibility and security. To achieve extensibility, REX bases much of its functionality on a single new abstraction—*emulated file descriptor passing across machines*. This abstraction is powerful enough for users to extend REX's functionality in many ways without changing the core software or protocol. REX addresses security in two ways. First, the implementation internally leverages file descriptor passing to split the server into several smaller programs, reducing both privileged and remotely exploitable code. Second, REX selectively delegates authority to processes running on remote machines that need to access other resources. The delegation mechanism lets users incrementally construct trust policies for remote machines. Measurements of the system demonstrate that the modularity of REX's architecture does not come at the cost of performance.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor, Department of Electrical Engineering and Computer Science, MIT

Thesis Supervisor: David Mazières
Title: Assistant Professor, Department of Computer Science, New York University

Acknowledgments

It goes without saying that I've come a long way since I began graduate school. I have many people to thank.

Frans Kaashoek and David Mazières co-advised this work and helped guide it to maturity. Frans has been a terrific mentor, always demanding the best. He has taught me much about computer science and research in general. Without David and the SFS project, many of the ideas in this thesis would never have come to fruition. Working with David has made me a better systems designer and builder.

I have collaborated with many people over the years, many of whom have contributed to this work in some way. In particular, I thank Chuck Blake, Kevin Fu, Daniel Giffin, Frans Kaashoek, Maxwell Krohn, David Mazières, Eric Peterson, and George Savvides. Daniel Jackson provided value feedback on the thesis document itself. Numerous reviewers read drafts of our SOSP and USENIX papers, offering many helpful comments. Being a member of PDOS has been a privilege; I have learned much during my time here.

MIT has been a tremendous environment for personal growth. I am indebted especially to the MIT Jewish community, and to the greater Boston area Jewish community, for providing me with a warm and caring environment. The friends that I've made over the years have taught me invaluable lessons about life.

It is difficult to express properly and adequately the thanks due one's parents. They provide the foundation upon which a person builds the rest of his life. My parents have done no less. Undeniably, I would not have made it to where I am today without their unwavering support and encouragement.

Several years ago, I would have concluded these acknowledgments with the previous paragraph. Since then, I have been blessed with an amazing wife and a beautiful family. Miriam deserves immeasurable gratitude that is hard to express in a few short sentences. She takes care of me, our children and our home. She is a source of strength and giving to our whole family. Thank you.

הודו לה′ כי טוב, כי לעולם חסדו!

This research was supported by the DARPA Composable High Assurance Trusted Systems program (BAA #01-24) under contract #N66001-01-1-8927; the National Science Foundation under Cooperative Agreement No. ANI-0225660 (as part of the IRIS Project); the National Science Foundation Graduate Research Fellowship; and the Alfred P. Sloan Research Fellowship.

Portions of this thesis are adapted from and/or contain text originally published in

> Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 60–73, Bolton Landing, New York, October 2003.

and

> Michael Kaminsky, Eric Peterson, Daniel B. Giffin, Kevin Fu, David Mazières, and M. Frans Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 199–212, Boston, Massachusetts, June 2004.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has changed the way people work. Increasingly, users are collaborating *across administrative domains* instead of just within them. Unfortunately, cross-domain collaboration is not as easy or as secure as collaboration within a single domain.

In a single domain, collaboration is both easy and secure because users have convenient systems for sharing and a trusted administrator that can make them secure. The two resources that users typically share are data (files) and processor cycles (computers). Network file systems allow users to share files, often with fine-grained access control, and remote execution and login utilities allow users to share machines. The trusted administrator can secure these systems in a number of ways. For example, he can place all of the computers that are on a private network behind a firewall, or he can set up a secure infrastructure for sharing over untrusted networks by distributing password files, public keys, and group lists.

Current systems for collaboration between administrative domains do not provide sharing that is both easy and secure. The two main systems for cross-domain collaboration are electronic mail (email) and the Web, which provide only application-level sharing rather than the generic file system and remote execution interfaces available within a single domain. The absence of a trusted, central administrator in cross-domain environments makes security challenging. The goal of this thesis is to make sharing resources across administrative boundaries both as easy and as secure as doing so within a single domain.

This chapter begins by providing a motivation for choosing a file system and a remote execution utility as the two basic mechanisms for sharing. It then discusses the problems, particularly related to security, that make extending these sharing mechanisms across administrative difficult. Sections 1.2 and 1.3 describe the two main contributions of this thesis that address these problems: a new user authentication system and remote execution utility. These sections also describe in more detail the motivation, technical challenges and approach for each system. Section 1.5 introduces our implementation environment, and Section 1.6 provides a road map for the rest of the thesis.

1.1 Sharing Across Administrative Domains

An *administrative domain* is a collection of users who trust an administrator to manage their computing infrastructure. Typically, domains represent an existing social structure. Examples of administrative domains include large organizations, such as universities, where the trusted administrator might be an IS department. Domains might be smaller, such as a research group, or even an individual with a single computer. The challenge in sharing across administrative domains is providing easy-to-use systems and interfaces while maintaining security even in the absence of a central, trusted administrator.

1.1.1 Current Approaches

Current systems for sharing across administrative domains are insufficient. For example, the two most popular ways to share files between organizations are email and the Web. Sharing documents over email has several disadvantages. Primarily, it does not provide a convenient interface for collaboration. Each email sent creates a new copy of the document, so keeping track of multiple versions quickly becomes unmanageable. Most email systems also do not provide security for cross-domain environments (i.e., no confidentiality and no authentication).

Publishing documents on secure Web servers (or Web portals) is also awkward. Working on a document might require several steps: logging into the system, downloading the latest version, modifying it, and uploading the modified document to the Web server. Most programs cannot perform these steps automatically and transparently; the user must open up a browser and perform them manually.

Both systems suffer from a common core problem: they provide convenient sharing interfaces perhaps for users but not for programs. Users prefer graphical, interactive interfaces that give all of the necessary context and require minimal effort to use. Programs, however, need narrow, well-defined interfaces that provide a basic set of low-level functions (e.g., read, write) and guarantees (e.g., consistency) with which the program can perform higher-order tasks. To be general-purpose, a system for accessing and sharing resources should provide "program-friendly" low-level interfaces to resources; then, using those low-level interfaces, applications can provide users with the high-level interfaces to the resources that they need.

In modern operating systems, most programs share through two basic program-friendly interfaces: the file system and stream abstractions (e.g., TCP sockets, terminals, pipes). These interfaces are also the most natural way for programs to access and share remote resources in other administrative domains. First, they provide a simple, proven interface that covers most of the sharing programs need. Second, existing programs already use these two interfaces, so sharing across domains is automatic and transparent.

Extending the two basic interfaces to work over the network is possible with two pieces of software—a network file system and a remote execution utility. The network file system server allows users to share files and directories that they own with other individuals. The file system client provides a way for those individuals to access the shared files remotely (without logging into the file server). The remote execution utility runs a program for the user on a remote machine, sometimes interactively (e.g., a login shell) and sometimes in the background. It provides a stream abstraction by effectively setting up a pipe between the remote program and a local one (which might be itself). The remote execution utility, through the stream abstraction, allows users to leverage computing resources that are available only on a different machine. Extending these two systems to work across administrative domains maintains the same program-friendly interfaces that are available in a single domain, but introduces new problems related to security.

1.1.2 Problem

The state-of-the-art network file systems and remote execution utilities lack a number of important properties necessary for use across administrative domains. With respect to network file systems, most existing systems were not designed for cross-domain sharing. NFS [10], for example, is insecure and inappropriate for wide-area networks. AFS [25] and the Microsoft Windows 2000 domain system [39] require an organized trust infrastructure based on pre-existing administrative relationships. SFS [38] overcomes these shortcomings by providing a secure, decentralized network file system with a global namespace; however, SFS has only basic support for user authentication across administrative domains. In particular, it does not support *cross-domain groups*—groups that contains users and other groups defined and maintained by administrators in other domains. For example, such a group might contain researchers, or even entire research groups, from several universities who are collaborating on a joint project.

SSH [65], the de-facto standard for remote execution, provides a secure stream I/O abstraction over the network, but it lacks several features important in cross-domain environments. First, the SSH architecture has limited extensibility. Collaboration across domains can introduce new needs and applications. Users need a modular way to add new functions and features without changing the protocols or even recompiling the software. Second, because the remote execution service is often accessible to the entire Internet and parts of the service must run with superuser privileges, the server software needs a clean design that limits remote exploits. Third, the system should allow restricted delegation of authority. In a single domain, users often trust all of the machines that they log into. When logging into machines in other administrative domains, where the remote machine might be less-trusted than local machines, users need the ability to restrict the authority they delegate to that machine. That is, the server should not be able to act on the user's behalf without the user's authorization.

1.1.3 Solution

This thesis presents two new systems that address the deficiencies in existing cross-domain network file systems and remote execution utilities. First, the thesis extends the original user authentication service currently available in SFS to support cross-domain groups [29]. These groups can contain users and nested sub-groups defined and maintained in other administrative domains. Second, this thesis introduces a new remote execution utility. It has an extensible framework for adding new features and functions outside of the core software and protocols; a safe, secure design that limits the amount of remotely exploitable code; and restricted delegation of authority to less-trusted machines [28].

The general approach taken in designing and building these systems is based on two principles. First, the systems should be convenient and practical. That is, the usage model should be familiar and intuitive. Creating cross-domain groups and sharing files should not be any more difficult than in the local case. The new remote execution utility should, from the user's point-of-view, operate similarly to existing tools. Second, the systems should have simple designs and implementations. If possible, the underlying abstractions should reflect those presented to the user. For example, the authentication service manipulates users and groups instead of certificates.

1.2 User Authentication

Several scenarios motivate the need for a new user authentication service that provides cross-domain groups. For example, a professor Charles at CMU wants to share a courseware directory with students in his class, as well as students at MIT and a user Liz at Boston University (BU). He creates a file sharing group that contains these users and groups, even though some of them are defined and administered in other domains. Then, Charles places this group on an Access Control List (ACL) for his courseware directory. The ACL lists the privileges associated with various users and groups (including the one he just created).

When a user accesses the directory, two critical steps must take place: user authentication and authorization. *User authentication* is the process of mapping a user to set of identities, which we call *credentials*. Typically, credentials include the user's name and the list of groups to which the user belongs. *Authorization* is the process through which a file server checks a user's credentials against an ACL to make an access control decision.

In the example above, when Liz at BU tries to access Charles's files, the user authentication service determines Liz's credentials: that she is Liz and that she is on Charles's file sharing group. The authentication service informs the file server of these credentials, and the file server makes an authorization decision. If Liz or the group she is in has privileges to perform her desired action, the file server permits her request.

1.2.1 Challenge

The primary task of the authentication service is to authenticate users who access the system. In the simple case, the user is defined locally and is known to the server through a password or a public key. Remote users defined in other domains, however, are unknown to the server, but they may in fact still merit access to a file or directory. The remote user might appear directly on an ACL or might appear as a member of a file sharing group that appears on an ACL.

Thus, for a user authentication service that operates across administrative boundaries, the main challenge is how to determine what groups the user is in. The key difficulty is that groups can contain users and even nested sub-groups that are defined in other domains. This level of indirection allows for complex group membership hierarchies, but it simultaneously complicates the user authentication process. Now, determining if a particular user is in a given group might involve entities in several different administrative domains.

1.2.2 Approach

There are a number of properties that one might want in a user authentication service:

1. **Flexibility**. The system should be able handle groups that span administrative boundaries, including nested groups.

2. **Bounded Staleness**. Group membership and user public key changes should take effect relatively soon. The system should bound the staleness of old authentication information.

3. **Scalability**. The system should scale to large numbers of users and groups.

4. **Simplicity**. The usage and administration, as well as the design and implementation, should be simple and intuitive.

5. **Privacy**. The system should ensure that group membership is private; furthermore, users should not have to reveal their identities.

Existing solutions make different tradeoffs in addressing these desirable properties. One popular approach to cross-domain user authentication, for example, is to create local accounts for all remote users that want to access local files. This solution, though simple, is not scalable and does not provide the flexibility desired. AFS [25] is an existing file system that supports groups and access control lists; AFS's authentication is based on Kerberos [55]. AFS provides a simple and intuitive file sharing interface, including a well-defined notion of freshness (e.g., Kerberos tickets expire regularly), but it does not work well across administration domains. Though one can set up cross-domain authentication in order to name remote users in specific domains, AFS does not support a notion of remotely administered groups (or even nested groups).

This thesis presents a new design point in the space of user authentication systems. The properties of this system are the flexibility to use remotely administered groups (including nested groups); a familiar, intuitive interface; and a simple design and implementation. The tradeoffs are that the system does not have perfect freshness (freshness where remote authentication information is verified each time it it used). The staleness of authentication information, however, can be bounded as described below. Experiments show that the system is scalable up to tens of thousands of users, groups, and users per group, though it might not scale to millions of users. The system also has limited privacy. Human-readable user names can be hidden behind public keys, but the list of which public keys belong to a group is public. Finally, the system provides all-or-nothing delegation. When a file owner names a remote group in his local group, he delegates to someone in the remote administrative domain the ability to add essentially anyone to his local group.

A new *authentication server* provides this authentication service. The authentication server has a familiar, intuitive 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 credentials for the user and returns them to the file server. 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 key design decision in this authentication system is to restrict ACLs to list only *local* users and groups. To name a remote user or group, the file owner creates a local group and places the remote user and/or group in that local group. He then places the local group on the ACL. The benefit of this design decision is simplicity because it allows for a clean separation between the authentication server and the file server. The file server requires only local knowledge; it does not need to understand anything about remote administrative domains. The authentication server can produce credentials without any additional information (such as the contents of ACLs) from the file server.

Another important design decision is that the authentication server pre-fetches and caches remote authentication information, so that authentication and authorization require only local data. This pre-fetching 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. This strategy trades off freshness for availability because the server makes authentication decisions with information that could be out-of-date. The advantage, however, is that the server does not need to block while contacting other machines at authentication time.

Though the authentication server does not provide perfect freshness, the server can bound the staleness to a fixed, configurable value (e.g., one hour). We do not believe that this compromise on freshness is an issue in practice. Many certificate-based systems make similar trade-offs; for example, certificates might still be usable for hours after revocation [63]. Some systems employ authentication caches that have timeouts [35, 4], and many systems use security tokens that specify group membership for as long as a user is logged in to his workstation [39].

1.3 Remote Execution

Remote execution and login are network utilities that many people need for their day-to-day computing; furthermore, many people layer their higher-level abstractions on top of remote execution utilities (e.g., CVS [1], *rsync* [2, 56]). Existing systems, however, do not provide all of the necessary features to operate adequately in today's complex network environments where users often move across administrative boundaries. These desirable features of a cross-domain remote execution utility fall into two categories: extensibility and security.

Extensibility. The concept of remote login is simple—local input is fed to a program on a remote machine, and the program's output is sent back to the local terminal. In practice, however, modern remote login utilities have become quite complex, offering their users many new features. For example, the popular SSH [65] program provides users with features such as TCP port and X Window System forwarding, facilities for copying files back and forth, cryptographic user authentication, integration with network file systems, transfer of user credentials across machines, pseudo-terminals and more. Extensibility means that adding new features should not require modifications to the core software or protocols; however, these new features must be added in a safe, secure way that does not increase the trusted code base.

Security. When running across administrative domains, security is particularly important. The remote execution utility should limit the amount of remotely exploitable code (code that deals directly with network connections) and the amount of code that runs with superuser privileges. The system should also provide restricted delegation of authority to processes running on remote machines. Delegation of authority allows a remote process to access and authenticate itself to remote resources. A user might want to limit that delegation if he trusts the remote machine less than the local one. Finally, the system should offer several ways to name remote servers securely, avoiding problems such as man-in-the-middle attacks.

1.3.1 Challenge

In remote execution, the main challenge is to design and build a remote execution utility that provides the diverse set of features necessary to operate across administrative domains. The utility should offer a convenient, practical and familiar usage model and a simple, extensible design that does not compromise security.

1.3.2 Approach

A new remote login and execution utility, called REX, operates across administrative domains. The main contribution of REX is its architecture centered around *file descriptor passing*, both as an internal implementation technique and as an external interface highly amenable to extensions.

REX provides extensibility through an abstraction that allows a process on one machine to effectively transfer a file descriptor to a process on another machine. In reality, REX emulates this operation by receiving the descriptor on one machine, passing a new socket to the recipient on the other machine, and subsequently relaying data back and forth between the descriptor and new socket over a cryptographically protected TCP connection. REX does not care if a passed file descriptor is the master side of a pseudo-terminal, a connection from an X-windows client, a forwarded connection to an agent that maintains the user's computing environment, or some as-yet-unanticipated future extension. Thus, REX's core system and protocol can provide the simplest possible interface on which external utilities can implement these more advanced features.

REX uses local Unix file descriptor passing to split the server into several smaller programs. This division of labor limits both the amount of code that runs with superuser privileges and the amount of code that deals directly with incoming network connections (which presents the greatest risk of being remotely exploitable). The REX server is split into two components: a small trusted program, *rexd*, and a slightly larger, unprivileged, per-user program *proxy*. Remote clients can communicate only with *rexd* until they prove that they are acting on behalf of an authorized user. *Proxy*, in turn, actually implements almost the entirety of what one would consider remote execution functionality. Almost all of the extensions and features in REX are implemented by this untrusted program (or some other untrusted helper program that *proxy* spawns). This separation of functions and privileges is possible because *rexd* uses local file descriptor passing to hand off incoming connections to *proxy*.

Another key security idea in REX is precise delegation of authority for users that need access to less-trusted servers. REX can prompt users to authorize remote accesses on a case-by-case basis. Users have an extensible, configurable “agent” process through which they can build up trust policies. By optionally instructing the agent to allow similar accesses in the future, users can construct these policies incrementally.

REX also offers users a number of different ways to name remote servers (corresponding to the server authentication mechanisms provided by the underlying secure computing infrastructure upon which REX is built) that do not rely on being susceptible to man-in-the-middle attacks. In SSH, for instance, the first time a user connects to a particular server, the server sends back its public key insecurely. The user must confirm that this key is the correct public key for the server. One technique REX uses to avoid man-in-the-middle attacks is to run the Secure Remote Password (SRP) protocol [62]. SRP allows the user and server to mutually authenticate each other based on a weak password. Additionally, REX can ask the user’s agent for the server’s public key; the user can configure his agent to provide this key by running an arbitrary program. Such a program might, for example, use a trusted corporate Lightweight Directory Access Protocol (LDAP) [58] server to obtain the key securely.

1.4 Contributions

This thesis presents three main contributions. The first contribution is a new design point for user authentication systems. The system supports groups that span administrative domains, and it has a simple design and implementation plus a familiar, intuitive user interface. The system trades off freshness for file system availability by having the server pre-fetch remote authentication information. It has scalability appropriate for file systems and offers limited privacy for group lists. Finally, the system has all-or-nothing delegation to other administrative domains via nested groups.

The second contribution is REX. REX introduces a new architecture for remote execution that allows users to operate more easily in the context of multiple administrative domains. The REX architecture is based largely on file descriptor passing, and it provides extensibility; a safe, secure design that limits exposure to exploits; precise delegation of authority; and sophisticated server naming mechanisms that avoid man-in-the-middle attacks.

The third contribution is an implementation of the authentication server and REX. Both systems were built in Self-Certifying File System (SFS) [38, 36] computing environment. The SFS infrastructure offers a convenient development platform on which to build these components; it already has a global file system with cryptographic server and user authentication, and it provides secure client-server network communication with which to build new services. Finally, SFS has built-in support for transparent access to servers without globally routed IP addresses (servers behind NAT)

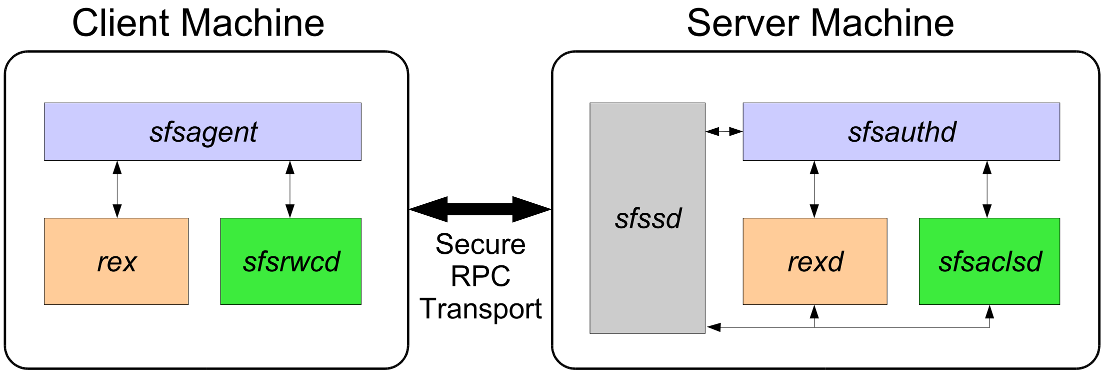


Figure 1-1: The SFS computing environment

and transparent connection resumption [28]. The following section gives an overview of the SFS computing environment.

1.5 The SFS Computing Environment

The user authentication server and remote execution utility (REX) are implemented as part of the SFS computing environment. This section gives an overview of the SFS architecture, summarizes the SFS security and trust model, and briefly describes how SFS provides transparent, persistent connections in the presence of NAT and dynamic IP addresses.

1.5.1 Architecture

SFS is a collection of clients and servers that provide several services—a global file system, remote execution, and user authentication. SFS clients and servers communicate using Sun Remote Procedure Calls (RPCs) [53]. The SFS implementation environment provides an RPC compiler and library, which promote security by offering a concisely-specifiable communication interface between local and remote components, and by parsing messages with mechanically-generated code.

When the client and server are located on different machines, the RPCs travel over a transport that provides confidentiality, integrity and authenticity [38]. This secure channel is set up using public-key cryptography as described below.

Figure 1-1 shows a single client and server in the SFS computing environment at a high-level. The machines are running two subsystems: a file system and REX. The file system client is called *sfsrwcd* and the file system server is called *sfsaclsd*. The REX client is called *rex* and the REX server is called *rexd*. These subsystems share a common authentication infrastructure made up of an agent (*sfsagent*) on the client machine and the authentication server (*sfsauthd*) on the server machine. An SFS meta-server (*sfssd*) receives incoming connections on a well-known port and hands those connections off to the appropriate sub-system.

1.5.2 Security and Trust Model

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. SFS does not prescribe any specific means for distributing server public keys to clients. A variety of 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. That is, the attacker can delay and/or truncate the message stream but cannot insert or change messages.

- **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. [38] describe the details of self-certifying hostnames and the protocols that SFS uses to set up secure connections. REX, *sfsauthd*, and *sfsaclsd* take advantage of these security properties offered by SFS.

1.5.3 Transparent, Persistent Network Connections

Two network configurations that users often encounter, particularly when working across administrative domains, are NAT and dynamic IP addresses. SFS has several mechanisms to provide transparent connection persistence in the presence of these network configurations. This section describes these mechanisms briefly; they are described more fully in previous work [28].

SFS employs two mechanisms to allow users to transparently connect to servers behind NAT boxes. First, the SFS meta-server, *sfssd*, can listen on a globally-routed (external) IP address for incoming connections. SFS connections all begin with a **CONNECT** RPC that includes the name of the server and service the user wants to access. Based on this RPC, *sfssd* can proxy requests to the appropriate server on the private network (similarly to the way that the HTTP `Host` header and HTTP proxies enable virtual Web servers). Second, SFS clients use DNS SRV records [24] to look up the IP address and port number associated with a service. In conjunction with a static TCP port mapping on the NAT box, SRV records can provide transparent access to machines without globally-routed IP addresses. For example, a user might want to access a service running on port 4 of `priv.sfs.net`, a machine on a private network behind a NAT box called `sfs.net`. The administrator of the NAT box configures it such that connections to port 9000, for example, are forwarded to port 4 of `priv.sfs.net`. Then, he sets up an SRV record for the service, which names the NAT box (i.e, `sfs.net` and port 9000). SFS clients that try to connect to this service will fetch the SRV record and connect to `sfs.net`, which will transparently forward the connection to `priv.sfs.net`.

SFS handles dynamic IP addresses and dropped TCP connections differently for its two main services. In the file system context, the underlying protocol is based on NFSv3 [10], which is designed to be stateless (on the server) and idempotent whenever possible. Thus, even if the network connection is lost, the client can resume where it left off once a new connection is established.

Remote execution connections, however, do not have these properties of statelessness and idempotence; therefore, REX provides transparent and automatic application-level session resumption. This feature allows REX to resume an interrupted remote login session, including any associated forwarding, over a new TCP connection.

1.6 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the design of a user authentication server, which supports groups that span administrative boundaries. It explains how the server authenticates users and determines the set of groups to which a user belongs. The chapter then discusses several properties of the system, such as freshness, performance, and scalability as well as certain optimizations that we have implemented. It concludes with a discussion of ACLs and authorization.

Chapter 3 describes the implementations of the authentication server (*sfsauthd*) and the SFS ACL-enabled file server (*sfsaclsd*). It provides a performance analysis of *sfsauthd* and an experimental evaluation of *sfsaclsd*. Finally, the chapter gives examples of how to use the system (*sfskey*, *sfsacl*).

Chapter 4 describes the REX architecture and connection setup, including definitions of file descriptor passing, sessions, and channels. The chapter then discusses the main goals of the REX architecture: extensibility and security. Chapter 5 offers an evaluation of the REX implementation in terms of code size and performance.

Chapter 6 discusses related work in the areas of network file system user authentication (specifically remote users and groups) and remote execution and login. Chapter 7 concludes with a summary and some open problems.

Chapter 2

User Authentication

This chapter describes a user authentication service that supports cross-domain groups in a file system context. The following example, introduced in Chapter 1, describes a typical file-sharing scenario and is used throughout the remainder of the chapter to illustrate the design of the authentication service. This scenario assumes the existence of a global, network file system that supports ACLs.

Charles is a professor at CMU who is teaching a course called CS100. He has developed courseware that he wants to share using a network file system. He places his courseware in a directory called `/home/cs100`. He creates entries on his ACL that describe the privileges he wants to assign to various users and groups. For example:

User/Group	**Privileges**
`u=charles`	read,write,admin
`g=charles.cs100-staff`	read,write
`g=charles.cs100-students`	read

These users and groups are defined in a local database at CMU. Users are simply public keys, and groups are lists of users and possibly other groups. For example:

User/Group	**Definition**	**Comment**
`u=charles`	`fg6aniv...`	Public key
`u=jenny`	`t7bwb8n...`	Public key
`u=joe`	`tf8kcnq...`	Public key
`g=charles.cs100-students`	`u=jenny`	Local user
	`u=joe`	Local user
	`...`	
	`g=math101-students`	Local group
	`u=liz@bu.edu,ur7vn2...`	Remote user
	`g=students@mit.edu,fr2eis...`	Remote grp
`...`		

Here, the group `charles.cs100-students` contains four different kinds of members: local users, local groups, remote users, and remote groups. Users are prefixed by `u=` and groups by `g=`.

Local users and groups are defined in CMU's authentication database; remote users and groups are defined in authentication databases in other administrative domains. Remote users and groups contain self-certifying hostnames (the server's DNS [42] name plus a cryptographic hash of its public key). All remote user and group names (those with an `@` character) are self-certifying, but for brevity the hash component of the names is omitted throughout many of the examples below.

When a user connects to the file server and tries to access Charles's courseware, the server must determine who the user is and what groups the user is in; that is, the server must generate credentials for the user. If the user is local, such as `jenny`, the server's task is straightforward. `Jenny` can prove she "is" a particular public key by signing a message using the corresponding private key. The server can verify the signature and look up her public key in its authentication database. Because `jenny` is local to CMU, an entry for her exists. Once the server knows it is talking to the user named "`jenny`", it can determine that `jenny` is a member of the `charles.cs100-students` group. Thus, the server can give `jenny` read access to the directory.

Authenticating the user accessing the file system is more complicated when the user is remote, such as `liz@bu.edu`. Liz can also prove she is a particular public key, but because she does not exist in CMU's authentication database, the server has no way to map her public key to a user name, even though her name does appear on Charles's group.

Finally, the user accessing the file system could be `alice@cam.ac.uk`, who does not appear on the group's membership list at all. Alice could, however, still merit access to Charles's files because the authentication service supports nested groups. `Alice@cam.ac.uk` could be a member of a group `mit-xchng@cam.ac.uk` (a group maintained by Cambridge University that contains a list of all of their students who are currently visiting MIT on an exchange program). The group `mit-xchng@cam.ac.uk` could be a member of the group `students@mit.edu` (a group maintained by the MIT registrar that contains all current MIT students). Charles, as shown above, placed `students@mit.edu` in his group `charles.cs100-students` at CMU. Thus, through a chain of nested group membership, `alice@cam.ac.uk` does actually merit access to Charles's courseware. This relationship, however, involves three different administrative domains; furthermore, the relationship might not be known to `alice@cam.ac.uk` ahead of time.

2.1 Architecture

A goal of the user authentication architecture (see Figure 1-1) is a simple design and implementation. To help achieve this goal, the system architecture separates the authentication server, which provides the user authentication service, from the resource servers (e.g., file servers, remote execution servers), which use that service. This separation allows the file server, for example, to have only local knowledge (i.e., ACLs can be restricted to contain only local users and groups), while the authentication server knows about other administrative domains. Having only local knowledge allows the authentication server to provide a generic user authentication service using a simple protocol as shown in Section 2.2.

The authentication server that a particular file server contacts to authenticate users is called the *local* authentication server. In the current implementation, the local authentication server runs on the same machine as the file server. This restriction, however, is not fundamental to the design of the system. Several file servers in a domain could share a single "local" authentication server 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.

2.2 Authentication Server

The authentication server serves two main functions. First, it provides a generic user authentication service to resource servers. When a user accesses a resource, the server asks the authentication server to determine the user's credentials. Once it has the credentials, the resource server can make an authorization (access control) decision.

Second, the authentication server provides an interface for users to manage the authentication name space. Users name remote authentication servers with self-certifying hostnames. They name remote users and groups that are defined on authentication servers in other administrative realms using the same idea (user name or group name "at" self-certifying hostname). Because remote user and group names are self-certifying, they are sufficient to establish a secure 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. Delegation is discussed in more detail at the end of Section 2.2.4.

2.2.1 Interface

The authentication server maintains a database of local users and groups. 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 a resource server to obtain a user's credentials (a user name and/or group list). The authentication server generates these credentials based on data from the client (this data is opaque to the resource server). **LOGIN** is the main step of the user authentication procedure described below (Section 2.3).

- **QUERY** allows a user (or another authentication server) to query the authentication database for a particular user or group record 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.

These RPCs always travel over a secure connection that provides confidentiality, integrity, and server authentication (or they are sent locally between two processes running on the same machine). **LOGIN**, by definition, does not require a previously *user*-authenticated connection. **QUERY** can be user-authenticated or not; when replying to an unauthenticated **QUERY**, the authentication server hides private portions of the user or group record (see Section 2.7). **UPDATE** requires a user-authenticated connection, so the authentication server can control access 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 resource server. The authentication server generates credentials for the user directly (it does not need to contact any external entity). Finally, the user issues **UPDATE** over the user-authenticated connection.

2.2.2 User Records

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

User Name
ID
GID
Version
Public Key
Privileges
SRP Information
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 [62]. 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.

2.2.3 Group Records

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

Group Name
ID
Version
Owners
Members
Properties
Audit String

Groups have a name Group Name. Groups created by regular users have names that are prefixed by the name of the user. For example, the user `charles` can create groups such as `charles.cs100-students` and `charles.colleagues`. Users with administrative privileges can create groups without this naming restriction. Each group also has a unique ID. Properties is a text field describing any additional properties of the group, such as refresh and timeout values (see Section 2.5.2).

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 user and group names, which are described below. Users who create personal groups implicitly own those groups (e.g., `charles` is always considered an owner of `charles.cs100-students`).

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.

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

```
p=bkfce6jdbmdbzfbct36qgvmpfwzs8exu
u=jenny
g=charles.cs100-students
u=liz@bu.edu,ur7bn28ths99hfpqnibfbdv3wqxqj8ap
g=students@mit.edu,fr2eisz3fifttrtvawhnygzk5k5jidiv
```

Figure 2-1: Example user and group names. Both `liz` and `students` are remote names, which include self-certifying hostnames.

2.2.4 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 SHA-1 hashes [16] of users' public keys. They are the most basic and direct way to name a user. User names refer to 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 key hashes, users, and groups, Owners and Members lists use the following two-character prefixes for each element: `p=`, `u=`, and `g=`. The table in Figure 2-1 shows several examples of these three types of names. (In the last example, `g=students@...` 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 do not need or want to run an authentication server. Second, they can provide a degree of privacy as described in Section 2.7.

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 2-1 shows how a user might name all of the students at MIT. The membership list for that group can be maintained by administrators at MIT, and people 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 the committee's two co-chairs. The owners and the members of this group all belong to different administrative organizations, but the authentication server provides a unified way to name each of them.

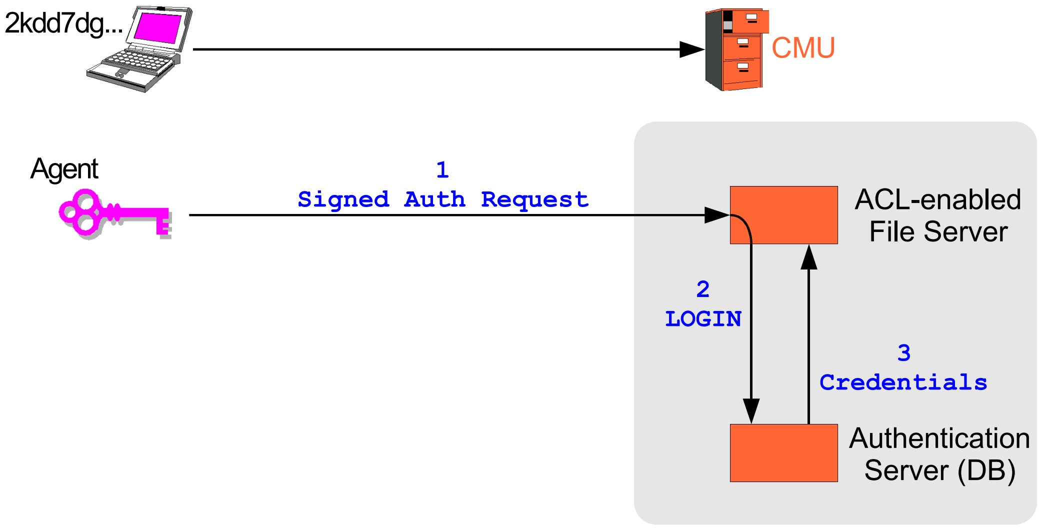


Figure 2-2: The user authentication procedure

Naming remote users and groups delegates trust to the remote authentication server. Delegation is important because it allows the remote group's owners to maintain the group's membership list, but it implies that the local server must trust those owners. A dishonest group owner could insert any number of users or groups into his group, even delegating further to another owner in a different administrative domain.

2.3 User Authentication Procedure

User authentication is the process through which users prove their identity to a server and get mapped to a set of credentials. The authentication service allows clients to initiate user authentication any time after the connection is set up. Typically, it 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).

User authentication is a multi-step operation that begins when a user accesses the file server. The user has a public-private key pair, which she stores in her agent. The agent is a per-user process running on the client machine that signs *authentication requests* on the user's behalf (see Figure 2-2). The user sends this signed request to the file server (Step 1), which passes it, as opaque data, on to the local authentication server using the **LOGIN** RPC (Step 2). 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 (Step 3), 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.

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 exist in the authentication database (i.e., registered users who have local accounts). Unix credentials play an important role in the default (non ACL-enabled) read-write file system, which uses NFSv3/Unix file system semantics for access control. REX also uses Unix credentials to set the user's shell and home directory. 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 a 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`.

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 next section describes how this is done.

2.4 Determining Group Membership

As illustrated at the beginning of this chapter, determining the groups to which the user belongs in a cross-domain setting is challenging. Because the design restricts ACLs to contain only *local* groups, the list of groups that the authentication server produces for the Group List credentials must also be local (defined on the authentication server itself). Any remote groups (or users) of interest to this authentication server will exist as members of some local group. Thus, the authentication server's task is to determine this list of local groups to which a user belongs.

The authentication server determines group membership for a user by performing a search starting from the local groups defined in its authentication database. For each local group, the server expands its membership list, which contains public key hashes, user names, and group names. For each member, the server uses the following rule:

- If the member is a public key or local user name, it stops.
- If the member is a local group name, it expands the local group, from the authentication database, and recurses.
- If the member is a remote user or group, the server contacts the remote authentication server securely using the self-certifying hostname contained in the user or group name. The server securely fetches the remote user or group's definition and then proceeds as in the local case. Fetching a remote user means fetching the user's public key; fetching a remote group means fetching the group's membership list.

The authentication server continues to expand users and groups in a breadth-first manner, building up a *containment graph*. The server avoids cycles by never visiting the same node twice. When

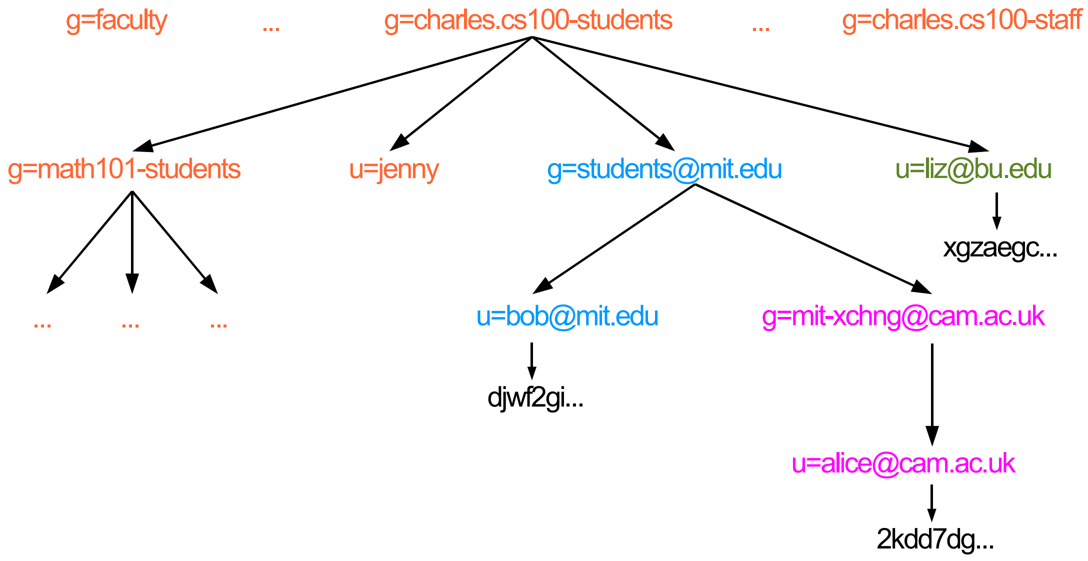


Figure 2-3: Containment graph for CMU

the traversal reaches a local user or a public-key hash, the server stops recursing. A path from a local group to a public key or local user name means that the user is a member of the given local group. The containment graph for the example given above (Charles at CMU) is shown in Figure 2-3. Containment graphs contain the expansions of *all* local groups defined at a particular authentication server, though the figure show only the expansion of `charles.cs100-students`.

Constructing the containment graph is easy for two reasons. First, the authentication server knows where to start its search (the local groups defined in its database). Second, the server knows how and where to find all of the information it needs. If the group is local, so is its definition. If the user or group is remote, the user name or group name is self-certifying, so the authentication server can fetch its definition from the appropriate remote machine securely.

Fetching remote user and group definitions when the user accesses the file system presents a problem because remote authentication servers might be unavailable or behind slow network links. In some cases, the number of remote authentication servers that the local server needs to contact can be quite large. Thus, constructing the containment graph on-demand might delay file access, which is often unacceptable.

2.5 Caching the Containment Graph

We address this problem by splitting the authentication task into two parts: constructing the containment graph and issuing credentials. The authentication server does the first part, constructing the graph, 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 containment graph cached, it can quickly generate the group membership list for the given user.

Pre-fetching and caching membership information in the background trades off freshness for availability, but it allows the authentication server to generate credentials *using only local informa-*

tion. 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.

2.5.1 Cache Entries

The cache contains an entry for each node in the containment graph that represents a remote user or group: remote users map to public key hashes and remote groups map to membership lists. Given the containment graph in Figure 2-3, the cache would consist of the following entries:

g=students@mit.edu:	u=bob@mit.edu
	g=mit-xchng@cam.ac.uk
	...

g=mit-xchng@cam.ac.uk:	u=alice@cam.ac.uk ...
u=liz@bu.edu:	xgzaegc...
u=bob@mit.edu:	djwf2gi...
u=alice@cam.ac.uk:	2kdd7dg...

The cache contains reverse mappings so the server can efficiently determine the group list for a user at authentication time. The server creates these extra pointers at the time it updates the cache. Each cache entry also has a refresh, timeout, and last updated value (see Section 2.5.2) and a version number (see Section 2.6).

2.5.2 Freshness

To update its cache of user and group records, the local authentication server simply rebuilds the containment graph. Updating the cache is a background task that the authentication server runs periodically. The frequency with which individual cache entries are updated determines their freshness. Although the system does not have perfect freshness, periodically updating the cache scheme bounds the staleness of cache entries.

Two factors influence a cache entry's update frequency: the local authentication server, which does the fetching, and the remote authentication server, on which the user or group is defined. The local authentication server's administrators can choose an update frequency that is appropriate for that system. This frequency might be based on how many groups need to be fetched in total and the machine's resource limitations. The remote authentication server's administrators can influence the update frequency because they know how often the user or group record changes. A group that only changes once a month does not need to be re-fetched every day.

The local authentication server influences the frequency with which cache entries are updated through a *global update frequency* value. This frequency determines how often the authentication server tries to rebuild its containment graph. The global update frequency defaults to once per hour, but administrators can configure it to a different value. A higher update frequency means that the authentication server will attempt to rebuild its containment graph more often.

Remote authentication servers influence the frequency with which cache entries are updated through *refresh and timeout values*. These values are similar to the refresh and expire parameters used in DNS [42] zone transfers. The owner of a user or group record (or an administrator) can set these optional refresh and timeout values (in the Privileges or Properties field), and the server returns them in response to a query. The refresh value indicates how often to fetch a new copy of the user or group record, and the timeout value indicates how long to keep using the cached copy of the record if the machine on which the record is defined is not available. If an entry does not have a timeout value and the remote machine is unavailable during the update cycle, the authentication server removes all references to that machine from the cache.

The global update frequency, per-record refresh, and per-record timeout values all play a role in determining the freshness properties of the system. Refresh and timeout values allow remote administrators to influence the freshness of cache entries, but these values are only suggestions. The local authentication server can ignore them if it wants to refresh or flush its cache entries more or less often. Regardless of the refresh value, the server will not update any cache entries more frequently than dictated by the global update frequency value (i.e., how often it rebuilds the containment graph).

In certain situations, the local authentication server will refresh one or more groups before the time dictated by the global update frequency, refresh, and timeout values. For example, when a user modifies a local group (e.g., by adding or removing members), the server will immediately update the cache entries related to that group. When necessary, administrators can also force an early cache update (for the whole server or just a particular group).

Although freshness is a key property of authentication systems, given the trade-off between availability (using only local information to issue credentials) and freshness, we chose availability. Delays during file system access are not acceptable. Given the trade-off between freshness and minimizing the time to rebuild the containment graph (update the cache), however, we chose freshness. Reducing the time required to complete an update cycle was not a priority because updating the cache is a background process. An earlier version of the authentication server sought to reduce the amount of time that updates took to complete by fetching records incrementally. That system, however, provided no guarantees on freshness.

2.5.3 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 (taking into account refresh and timeout values). If the administrator of the local authentication server learns about the revocation, he can manually update his server's cache before the normal update cycle takes place. Updating the cache, however, does not revoke existing connections because users only authenticate when they first log into the file system.

For users who have their public key hashes on another user's group record or ACL (see Section 2.8.1), 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 7.2 suggest several possible solutions.

2.6 Optimizations

The authentication server implements three performance optimizations. First, during each update cycle, the local authentication server connects only once to each remote authentication server and maintains that open connection for later use. Caching connections to authentication servers avoids the public-key operations involved in establishing new secure connections. For a large number of remote users or a large remote group, the savings can be significant. (If the underlying transport used SSL-like session resumption [14, 19], the authentication server would not need to implement this optimization manually. This style of connection resumption would have the benefit of avoiding repeated public-key cryptography operations without maintaining open TCP connections.)

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.) Section 7.2 suggests an alternative approach that would allow the local server to fetch group lists from a closer replica instead of the authoritative server on which they are defined. Fetching from replicas can help reduce load on authentication servers with popular groups.

Third, remote authentication servers have a configuration option that causes them to transform local user names (local to that 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 `mit-xchng@cam.ac.uk`, the remote authentication server `cam.ac.uk` would return the public key `2kdd7dg`...instead of the user name `alice@cam.ac.uk`. The server `cam.ac.uk` knows (or can quickly compute) `2kdd7dg`...because it has the user record for `alice@cam.ac.uk` 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. Because group owners want to see user names when they manage their groups, the authentication server does not perform this transformation for user-authenticated **QUERY** RPCs. (**QUERY** RPCs from other authentication servers travel over a secure connection that provides server authentication but not user authentication; that is, the client authentication server is anonymous.)

2.6.1 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 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.

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.

2.6.2 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. 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. Naming all of the citizens of a country or all Visa credit card holders might require a different technique.

For example, MIT's Athena setup (which is based on Kerberos and AFS) has 19 file servers holding user data, and a total of 20,363 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 authentication server is well-equipped to handle authentication for a large institution, as demonstrated in Section 3.1.1.

2.7 Privacy

A consequence of having the server determine group membership through the containment graph is that group lists must be publicly available. Sometimes these group lists can contain private information, such as the names of individuals who are members of a particular interest group or the organizational hierarchy of a company's employees. If the group list contains user names that correspond to email addresses, those addresses might be abused to send spam.

Though the ability to fetch a group list is fundamental to the design of the system, the optimization described in Section 2.6 provides some degree of privacy. By returning public keys instead of local user names, administrators obfuscate that part of a group's membership list.

The current authentication server does not provide any way to hide local group names or remote user and group names. Such privacy-enhancing mechanisms might be possible, to some extent, as follows. To hide local group names, the server could similarly transform the group name into a meaningless, arbitrary identifier. When it receives a query for that identifier, the server simply returns the contents of the original group. This technique reveals the structure of the group membership hierarchy, but not the members of the group list.

Obfuscating remote user and group names is more difficult. One option might be for the owner of the user or group to give it a meaningless identifier, so that others can refer to the group without revealing its purpose. This technique, however, does not hide the self-certifying portion of the remote user or group name, which might contain a meaningful DNS name; furthermore, using meaningless, arbitrary names severely reduces usability.

In general, obfuscating user names with public keys does not prevent directed attacks, in which an adversary knows the public key of a particular individual and then checks it against specific group lists, or a correlation attack, in which an adversary compares two group lists to learn if any public keys (users) are members of both groups. In the latter case, one possible solution is to use different

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

keys on each group list. Unfortunately, this approach also reduces usability because one must know in advance which key to use when authenticating to a server; moreover, guaranteeing a different key on each group list is difficult because the owner of the key (the user) is typically not the person creating the group.

Providing privacy in this style of user authentication service, where the server is responsible for fetching group lists, is an open area for research. Section 7.2 discusses these privacy issues further and some of the tradeoffs that exist.

2.8 A File System with ACLs

Once the user has credentials, the various resource servers (e.g., the file system, REX) can make access control decisions based on those credentials. For example, the SFS computing environment has a read-write file system that 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.

A modified version of the this read-write file system supports access control lists. In the ACL-enabled variant of the 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 ACL-enabled file system 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 (e.g., a group list) 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 [23]. These new file systems might provide an alternative to the ACL-enabled file system. To allow the operating system to make access control decisions, however, the authentication server would need to ensure that each of its local user and group records had a corresponding entry in `/etc/passwd` and `/etc/group`.

2.8.1 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. In the SFS-based ACL-enabled file system, 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 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 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 define personal groups (prefixed by their user names) on the authentication server and place remote user and group

Permission	Effect on files	Effect on directories
r	read the file	no effect, but inherited by new files
w	write the file	no effect, but inherited by new files
l	no effect	enter 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

Table 2.1: Access rights available in ACLs

names on the membership lists of those new groups. This restriction offers several benefits. Primarily, it allows the server to pre-fetch all remote authentication information. Adding a level of indirection also 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.

2.8.2 Access Rights

We adopted the AFS [25] ACL access rights but extended them to differentiate between files and directories (AFS only has ACLs on directories). Table 2.1 lists the different permissions that an ACL can contain and the meaning that each permission has. Unlike AFS, the ACL-enabled file server does not support negative permissions; once an ACL entry grants a right to the user, another entry cannot revoke that right.

Chapter 3

User Authentication: Implementation, Evaluation and Usage

This chapter evaluates the authentication server and ACL-enabled file server described above. The evaluation of the authentication server is based on how many bytes a server needs to transfer when fetching groups. We expect the number of bytes to scale with group size and to be such that the system can scale to tens of thousands of members per group. The evaluation of the ACL-enabled file server measures the performance penalty of reading, writing, and processing ACLs. We expect, with appropriate caching, that the ACL-enabled file server will exhibit a minimal slowdown with respect to the standard SFS file system. The chapter concludes with an example of how one might use the system.

3.1 Authentication Server

The implementation of the authentication server in the SFS computing environment is called *sfsauthd*. It is fully functional and supports cross-domain, nested groups. *Sfsauthd* implements an RPC interface (described in Section 2.2.1) to its authentication database. To improve scalability, the server has a Berkeley DB [5] backend, which allows it to efficiently store and query large groups. 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.

3.1.1 Evaluation: Methodology

Because the cache update scheme is a background process, the important metric by which to evaluate the authentication server's update scheme is how many bytes must be transferred (the computation that the server must do is insignificant). The number of bytes that the 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

		To Transfer			
		0 **users**	**10,000** **users**	**0** **changes**	**1,000** **changes**
Size of RPC Request (bytes)	Q	72	72	72	72
Size of RPC Reply (bytes)	R	136	136	108	108
Size of Single User/Change (bytes)	S	40	40	40	40
Number of Users/Changes	M	0	10,000	0	1,000
RPC Overhead Per 250 Users (bytes)	O	216	216	180	180
Total Bytes Transferred	B	208	408,632	180	40,720

Table 3.1: Number of bytes transferred when fetching group records

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 connection. Because the implementation caches connections, it establishes only one such connection per update cycle per remote server; 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.

3.1.2 Evaluation: Results

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).

Table 3.1 shows sample data from these experiments. Q is the size of the RPC request, R is the size of the reply (excluding the group’s users/changes), 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 ($O \approx Q+R$). B is the total number of bytes transferred. All values (except for M) are in bytes.

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:

$$B = Q + R + (M \times S) + max\left(\left\lceil \frac{M}{250} \right\rceil - 1, 0\right) \times O$$

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 10,000 users requires approximately 400 KB. The total RPC overhead is only $\lfloor \frac{M}{251} \rfloor \times O = 8,424$ 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 1,610 members. Based on the formula above, the number of bytes required to transfer that group is 65,904.

As a thought experiment, if all of the Athena groups existed as groups in an *sfsauthd* database, and a different authentication server (in a different domain) wanted to fetch *all* 20,000 plus groups, the transfer would require just under 16 MB. Once the groups are cached, subsequent runs of the update cycle would transfer approximately 3.5 MB (= 180 bytes for zero changes × 20,363 groups). This result, however, represents a worst-case scenario. Most likely, a remote administrative domain would not want to reference all of these groups, most of which are not relevant outside of MIT. In fact, about one half of the 20,363 groups contain only a single user. If such scenarios became common, new RPCs could be introduced that first retrieve a list of groups that have changed, for example.

The total number of bytes that an authentication server must transfer during each update cycle depends on the size of the entire containment graph, not just a single group or a set of groups from a single remote administrative domain. Because delegation through nested sub-groups is easy, these graphs could become quite large in theory. Future usage studies and deployment experience might indicate how people would build group hierarchies in practice.

3.2 ACL-Enabled File System

The SFS ACL-enabled file system (*sfsaclsd*) [51] is an extension of the SFS read-write file system. Both variants store files on the server's disk using NFSv3 [10]. This technique offers portability to any operating system that supports NFSv3 and avoids the need to implement a new in-kernel file system.

3.2.1 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. *Sfsaclsd* stores file ACLs in the first 512 bytes of the file and directory ACLs in a special file in the directory called `.SFSACL`.

These 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. NFS requests contain *file handles* and storing the ACLs in a database indexed by NFS file handle would technically be possible; however, it would impose constraints on how one treated the underlying file system. 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, *sfsaclsd* 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, *sfsaclsd* 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.

3.2.2 ACL Format

For rapid prototyping and to simplify debugging, we chose to use a text-based format for the ACLs. The ACL for Charles's `/home/cs100` directory described in the previous chapter might be:

```
ACLBEGIN
user:charles:rwlida:
group:charles.cs100-staff:rwlid:
group:charles.cs100-students:rl:
ACLEND
```

Charles creates these groups on his local authentication server before placing them on the directory's ACL. Permissions such as `r` and `w` have no effect on directories, but they are still useful 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.

3.2.3 Permissions

When *sfsaclsd* 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 file system client can determine these rights (for instance, when opening a file) by issuing the NFSv3 **ACCESS** RPC. The ACL-enabled file server replies to **ACCESS** based on the object in question's ACL.

File attributes returned by the server still contain standard Unix permission bits. These bits are not used by the file server to determine whether the request will succeed, but they may be useful to the client. The 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.

3.2.4 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 can 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).

3.2.5 Evaluation: Methodology

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

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

Table 3.2: LFS small file benchmark, with 1,000 files created, read, and deleted. The slowdowns are relative to the performance of the original SFS.

needs to issue in order to locate and read (or write) the ACLs associated with the incoming requests from the client.

To quantify the penalty, 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 [49] 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 client kernel's buffer cache, but we set the sizes of the internal ACL and permission caches to large enough values so that entries were not flushed from those 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.

3.2.6 Evaluation: Results

Table 3.2 shows the results of running the benchmark on the three file system variants. The figure breaks down the best result of five trials.

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. In particular, the NFS **WRITE** RPC to write the ACL itself is synchronous (the server commits all of the ACL data to stable storage before returning). Processing the directory's 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.

Table 3.3 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.

The last row of Table 3.3 lists predicted slowdown, the ratio of the number of NFS RPCs required by the ACL-enabled file server (with and without caching) to the number required by the original SFS file server. With caching enabled, the actual read-phase performance slowdown (1.03×)

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	1.00×	1.00×	2.67×

Table 3.3: 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.

basically agreed with the predicted slowdown (1.00×). The difference is likely due to the overhead of cache lookups.

With caching disabled, the actual slowdown (1.26×) was much lower than the predicted slowdown (2.67×). 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 1.02×.

3.3 Usage

We provide two tools for manipulating groups and ACLs. Examples of their usage are given below. First, we've extended the *sfskey* utility to view, create, and update group lists on an authentication server.

Second, a new utility, *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.

The following example demonstrates how Charles at CMU might use this system to share his course software. The software is on Charles's ACL-enabled file server in a directory called `/home/cs100`. First, Charles creates a personal group on the authentication server:

```
$ sfskey group -C charles.cs100-students
```

He can now add members to this new group:

```
$ sfskey group \
   -m +u=jenny \
   -m +u=liz@bu.edu,ur7bn28ths99hfpqnibfbdv3wqxqj8ap \
   -m +g=students@mit.edu,fr2eisz3fifttrtvawhnygzk5k5jidiv \
   -m +p=anb726muxau6phtk3zu3nq4n463mwn9a \
  charles.cs100-students
```

`jenny` 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.

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

```
$ sfskey group \
   -o +u=george@sun.com,ytzh5beann4tiy5aeic8xvjce638k2yd \
  charles.cs100-students
```

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.cs100-staff:rwlid:
group:charles.cs100-students:rl:
ACLEND
$ sfsacl -s myacl.txt /home/cs100
```

Charles has full access permissions to the directory, his course staff have read-write access, but the members of `charles.cs100-students` can only read and list its contents.

Chapter 4

REX Design

This chapter presents REX, a remote execution utility designed to work across administrative domains. REX has an extensible, modular architecture that does not require changes to the core protocol and software in order to add new features. REX provides this extensibility without compromising security, which is particularly important when operating across administrative domains.

Section 4.1 begins with an overview of the architecture, and Section 4.2 discusses REX session establishment. Sections 4.3 and 4.4 describe how REX achieves its main goals of extensibility and security.

4.1 Architecture

At its most basic level, REX provides a secure pipe abstraction between two machines. This secure pipe is called a REX *session*, and it typically corresponds to a single TCP connection between a REX client and a particular server. Sessions are the mechanism through which users run programs on the server. Section 4.2 describes how REX establishes new sessions. This section describes the anatomy of a session and provides a simple example of how remote execution works in REX. It concludes with a discussion of file descriptor passing, the basic building block for extensibility in REX.

4.1.1 Sessions, Channels, and Remote Execution

Figure 4-1 shows the anatomy of a REX session. A REX session is a connection between a process called *rex*[1] running on the client machine and a process call *proxy* running on the server (Section 4.2 provides details on how sessions are established). Each session contains one or more REX *channels*. A channel is an abstraction that connects a pair of programs running on different machines; these programs are called *modules*. REX channels allow modules to communicate as if they were running on the same machine, connected by one or more Unix-domain socket pairs. The abstraction within a channel that represents this socket pair is called a REX *file descriptor*.

Users run programs on the server by creating new channels. The channel protocol involves three main RPCs, summarized in Table 4.1 and described more fully below. The *rex* client creates a new channel by sending a **REX_MKCHANNEL** RPC to *proxy*. The RPC specifies the name of the server module to run, a set of command-line arguments and environment variables to set, and the number of file descriptors the spawned module should inherit. (If fewer than three file descriptors

[1] This thesis uses REX (capital letters) to refer to the remote execution utility as a whole and *rex* (italicized lowercase) to refer to the client program that the user invokes to start a REX session.

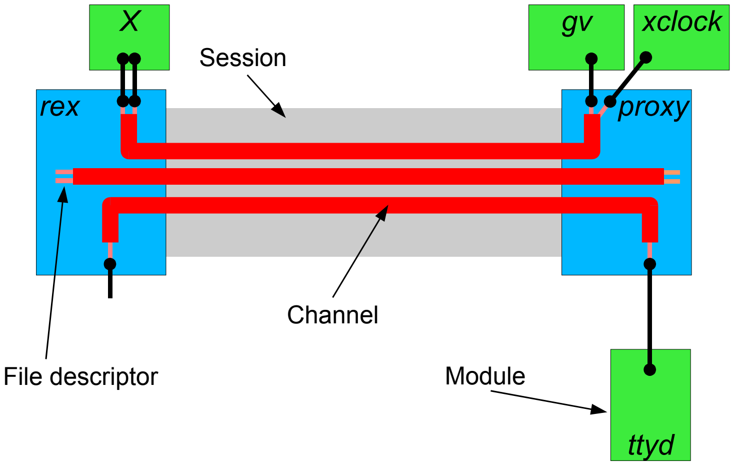


Figure 4-1: Anatomy of a REX session

RPC	Direction	Description
REX_MKCHANNEL	*rex* → *proxy*	Create a new channel
REX_DATA	*rex* → *proxy*	Send data over a channel/file descriptor
REX_NEWFD	*rex* → *proxy*	Pass a new file descriptor
REXCB_DATA	*proxy* → *rex*	Send data over a channel/file descriptor
REXCB_NEWFD	*proxy* → *rex*	Pass a new file descriptor

Table 4.1: REX channel protocol RPCs

are specified, standard input, standard output, and possibly standard error of the spawned process will be the same socket.) Depending on the channel, *rex* can either redirect I/O to a local module, or else relay data between the channel file descriptors and its own standard input, output, and error. Figure 4-1 shows several types of channels, including one for X11 forwarding and one for TTY support. These channels are described in Section 4.3.

Figure 4-2 illustrates how a user might execute the program *ls* on the server using REX. The user runs the command `rex -x host ls`. (The `-x` argument disables X11 forwarding.) The *rex* client sends a **REX_MKCHANNEL** RPC to *proxy* specifying that the user wants to run *ls* and that three REX file descriptors should be set up within the channel. When *proxy* receives this RPC, it creates a new channel within the session and spawns *ls* as the server module. *Proxy* assigns a unique channel number (e.g., 15) to the channel and returns that number to *rex*.

The *rex* client, in this example, acts as the client module itself (instead of spawning a separate program). *Rex* relays data that appears on the three REX file descriptors in the channel to its own Unix standard file descriptors (FDs 0, 1, and 2). On the server, *proxy* spawns *ls* and creates three Unix-domain socket pairs. *Proxy* connects one end of each socket pair to *ls*'s standard file descriptors, and it keeps the other end of these Unix-domain socket pairs itself, relaying data between them and the REX file descriptors in the channel.

Figure 4-3 shows how data travels through the REX channel when *ls* produces output. First, *ls* writes to its own standard output (FD 1). That data travels through the Unix-domain socket pair that *ls* shares with *proxy*. *Proxy* reads the data from its end of the socket pair and then generates

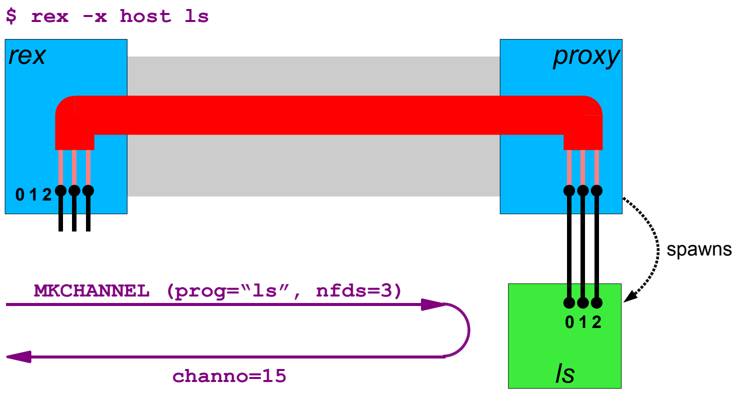


Figure 4-2: Using a REX channel to run *ls*

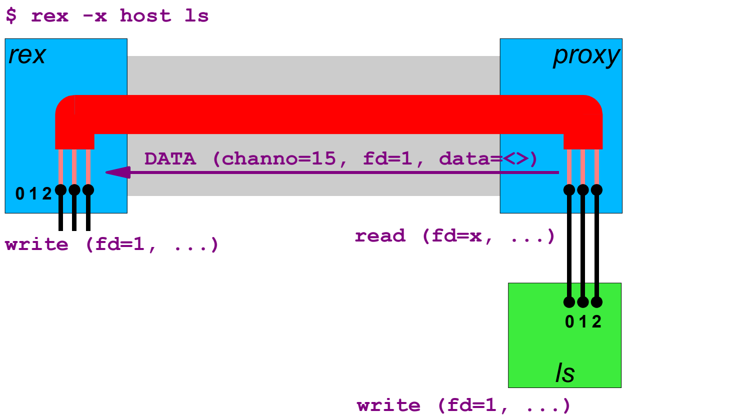


Figure 4-3: How data travels through REX when *ls* produces output

a **REXCB_DATA** RPC to *rex*, specifying the REX channel number and file descriptor number on which the given data appeared. When *rex* receives this RPC, it copies the data from the indicated channel file descriptor to the appropriate Unix file descriptor (in this case, FD 1). If the *rex* client's standard output is the user's terminal, for example, the data from *ls* will appear there.

4.1.2 File Descriptor Passing

In addition to replicating reads and writes across the network, REX channels emulate *file descriptor passing* between modules running on different machines. Unix provides a facility for file descriptor passing between two processes running on a single machine that are connected by a Unix-domain socket pair [45]. Using the *sendmsg* and *recvmsg* system calls, one process can send one of its file descriptors to the other process. The receiving process effectively gets a copy of the sending machine's file descriptor (i.e., both sides have a file descriptor that refers to the same kernel data structure). File descriptor passing is like the *dup* system call, except across processes.

The channel abstraction is the mechanism through which REX emulates file descriptor passing over the network. When a client module passes a file descriptor to *rex*, *rex* notifies *proxy* through the **REX_NEWFD** RPC. *Proxy* then creates a new Unix-domain socket pair, passes one end to

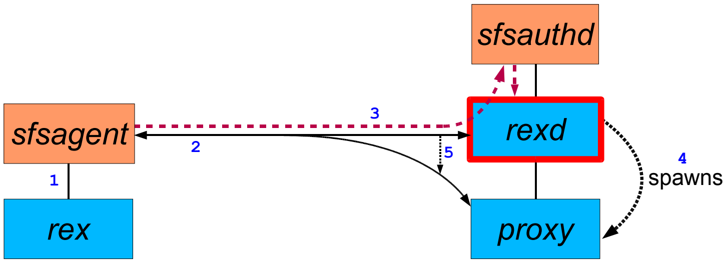


Figure 4-4: Setting up a REX session (Stage I). *Rexd* runs with superuser privileges (shown with a thick border).

the local server module, and allocates a new REX file descriptor number within the channel for the other end. Conversely, when a server module passes a file descriptor to *proxy*, *proxy* allocates a new REX file descriptor number for it within the appropriate channel and notifies *rex* through the **REXCB_NEWFD** RPC. *Rex* makes a new socket pair and passes one end to the local client module. As Section 4.3 demonstrates in detail, this emulated file descriptor passing is the foundation of REX's extensibility.

4.2 Establishing a Session

REX sessions provide a secure pipe between two machines. All subsequent communication is layered on top of this pipe. A primary goal in REX session establishment is security. Only the code that needs superuser privileges has those privileges. A secondary goal in REX session establishment is efficiency. Remote execution should be cheap, particularly if running multiple programs on the same machine.

Establishing a REX session has two stages. In Stage I, the user's agent establishes a secure, authenticated connection to the server using public key cryptography. This initial connection by the agent is called the *master* REX session. In Stage II, the REX client creates new REX sessions, based on the master session, to run programs on the server. When the user invokes the *rex* client, the client first checks with the user's agent to see if a master session already exists with the desired server. If so, *rex* proceeds directly to Stage II; otherwise, *rex* continues with Stage I and sets up a new master REX session.

4.2.1 Stage I

Rex begins by contacting the *sfsagent* and asking it to establish a new master session to the desired server (Figure 4-4, Step 1). In Step 2, the *sfsagent* uses the server's public key to establish a secure connection to the *rexd* process running on the server.[2] Section 4.4.3 describes several mechanisms through which the client can obtain the server's key.

Next, the *sfsagent* authenticates its user to *rexd* (Step 3). The authentication procedure that REX uses is the same one described in Section 2.3. The agent signs an authentication request, which it passes to the server through the secure connection. *Rexd* passes the authentication request to the

[2] By default, the agent's connection to *rexd* goes through the *sfssd* "meta-server" as described in Section 1.5.1; for simplicity, this extra step is omitted in here.

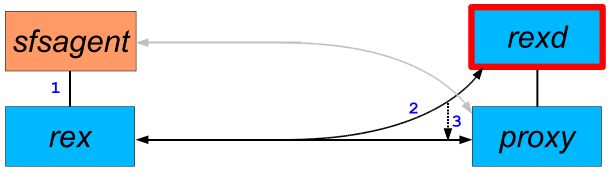


Figure 4-5: Setting up a REX session (Stage II). The gray line represents the master REX session established during Stage I.

authentication server, *sfsauthd*, which verifies the signature and produces credentials for the user. *Rexd* uses the Unix credentials it receives to map the client user to a local account.

Once the user is authenticated, *rexd*, which runs with superuser privileges, spawns a new process called *proxy*, which runs with the privileges of the local user identified above (Step 4). *Proxy* is responsible for most of the functions normally associated with remote execution.

The *sfsagent* and *rexd* now generate new symmetric cryptographic keys (one for each direction). These keys are known as the *MasterSessionKey*s. All subsequent sessions with this *proxy* process use session keys that are derived from these *MasterSessionKey*s.

Rexd hands *proxy* the connection it has with the *sfsagent* (Step 5). The *sfsagent* and *proxy* now communicate with each other directly over their own connection, using their own protocol and their own cryptographic keys (derived from the *MasterSessionKey*s). This connection is called the master REX session. *Rexd* keeps track of all master REX sessions (i.e., all *proxy* processes it has spawned). The *sfsagent* maintains its connection to *proxy* in order to keep the master session alive; once the agent closes its connection to *proxy* (provided no other clients are still connected), *proxy* will exit and *rexd* will delete the master session.

4.2.2 Stage II

The master REX session is a connection between the user's *sfsagent* and *proxy*, but all subsequent sessions are connections between *rex* and *proxy*. To establish such a session, the *rex* client notifies the *sfsagent* that it wants to create a new session to the desired server (Figure 4-5, Step 1).

The *sfsagent* computes the values shown in Figure 4-6 based on the *MasterSessionKey*s that were established when *proxy* was spawned. The *SessionKey*s are the symmetric keys that the *rex* client will use to encrypt its connection to *proxy*. They are computed as the HMAC-SHA-1 [16, 33] of a sequence number *i* keyed by the *MasterSessionKey*s. The agent generates a unique sequence number for each new REX session to prevent an adversary from replaying old sessions. The *SessionID* is a SHA-1 [16] hash of the *SessionKey*s, and the *MasterSessionID* is the *SessionID* where the sequence number is 0.

Once the *sfsagent* computes these values, it returns them to the *rex* client. *Rex* makes an initially insecure connection to *rexd* (Step 2) and sends the sequence number, the *MasterSessionID*, and the *SessionID*. Session IDs can safely be sent over an unencrypted connection because adversaries cannot derive session keys from them. *Rexd* looks up the appropriate master session (i.e., the appropriate *proxy* process) based on the *MasterSessionID*. Then, *rexd* computes the *SessionKey*s and the *SessionID* for the new REX session (as in Figure 4-6) based on the sequence number that it just received and the *MasterSessionKey*s that it knows from the initial connection by the *sfsagent*.

$$\begin{aligned}
SessionKeySC_i &= \text{HMAC-SHA-1}(MasterSessionKeySC, i)\\
SessionKeyCS_i &= \text{HMAC-SHA-1}(MasterSessionKeyCS, i)\\
SessionID_i &= \text{SHA-1}(SessionKeySC_i, SessionKeyCS_i)\\
MasterSessionID &= SessionID_0
\end{aligned}$$

Figure 4-6: *Sfsagent* and *rexd* use the *MasterSessionKey*s and sequence number (*i*) to compute new *SessionKey*s.

Rexd verifies that the newly computed *SessionID* matches the one received from the *rex* client. If they match, *rexd* passes the connection to *proxy* along with the new *SessionKey*s (Step 3). Finally, *rex* and *proxy* both begin securing (encryption and message authentication code) the connection.

4.2.3 Connection Caching

The protocol REX uses to set up subsequent sessions (Stage II) is a form of *connection caching* [14, 20]. *Rex* uses the *sfsagent* to establish a master session with *rexd*/*proxy* first. This initial REX connection is set up using public-key cryptography. Once this connection is established, REX uses symmetric cryptography to secure communication over the untrusted network. The *sfsagent* remembers (maintains) this connection in order to set up subsequent REX sessions to the same machine quickly. Subsequent sessions bypass the public-key step and immediately begin encrypting the connection using symmetric cryptography.

For an interactive remote terminal session, the extra time required for the public-key cryptography might go unnoticed, but for batched remote execution that might involve tens or even hundreds of logins, the delay is observable. Connection caching offers an added benefit; if the user's agent was forwarded, that forwarding can remain in place even after the user logs out, allowing him to leave programs running that require use of the his *sfsagent*. *Sfskey* lets the user list and manage open connections.

4.3 Extensibility

One of the main design goals for REX is extensibility. SSH has demonstrated that users want more features than just the ability to execute programs on a remote machine. TTY support, X11 forwarding, port forwarding, and agent forwarding, for example, are critical parts of today's remote execution tool. When working across administrative domains, however, the need can arise for new, unanticipated features. REX, therefore, not only offers the features found in existing remote execution utilities, but it also provides users with a simple, modular interface to add new ones. REX's extensibility stems primarily from the REX channel's ability to emulate file descriptor passing over the network. None of the features described in this section required any changes to the REX protocol.

4.3.1 TTY Support

REX provides optional pseudo-terminal support to interactive login sessions (allocating a pseudo-terminal is the default behavior if no command-line options are specified). REX implements pseudo-terminal support using the channel abstraction and file descriptor passing as follows. When the user

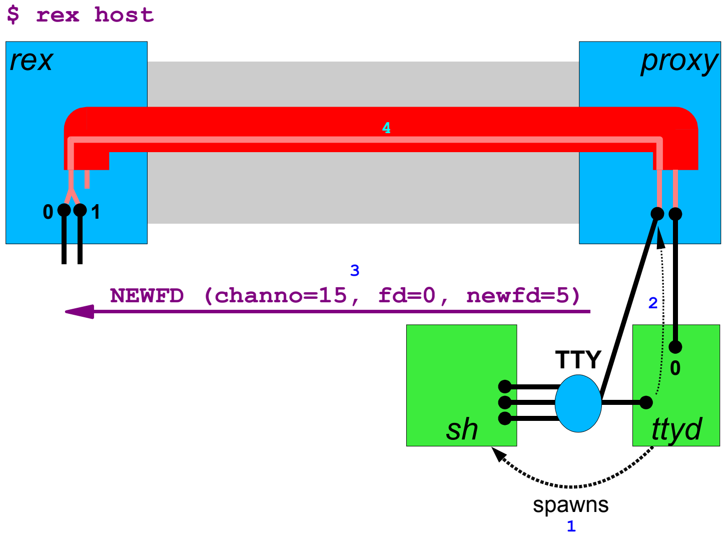


Figure 4-7: Supporting TTYs with emulated file descriptor passing

runs `rex host`, REX establishes a new session. Then, the *rex* client sends a **REX_MKCHANNEL** RPC to *proxy* telling it to launch a module called *ttyd*. *Ttyd* takes as an argument the name of the actual program that the user wants to run. For an interactive login session, the user can omit the argument to *ttyd*, and *ttyd* will automatically default to the user's shell. *Proxy* spawns *ttyd*, setting up a Unix-domain socket pair between them as in the *ls* example above (here, however, the channel has only one initial file descriptor).

Ttyd runs with only the privileges of the user who wants a TTY. The program has two tasks. First, it obtains a TTY from a separate daemon running on the server called *ptyd*. *Ptyd* runs with superuser privileges and is responsible only for allocating new TTYs and recording TTY usage in the system `utmp` file. The two processes, *ttyd* and *ptyd*, communicate via RPC. When *ptyd* receives a request for a TTY, it uses file descriptor passing plus an RPC reply to return the master and slave sides of the TTY. *Ttyd* connects to *ptyd* with *suidconnect*, SFS's authenticated IPC mechanism (described further in Section 4.3.4). This mechanism lets *ptyd* securely track and record which users own which TTYs.[3] After receiving the TTY, *ttyd* keeps its connection open to *ptyd*. Thus, when *ttyd* exits, *ptyd* detects the event by an end-of-file. *Ptyd* then cleans up device ownership and `utmp` entries for any TTYs belonging to the terminated *ttyd*.

Once *ttyd* receives a newly allocated TTY, its second task is to spawn the program given as its argument, or, more commonly, the user's shell (see Figure 4-7, Step 1). *Ttyd* spawns the shell with the slave side of the newly allocated TTY (left) as the shell's standard file descriptors and controlling terminal. Then, *ttyd* uses local Unix file descriptor passing to send the file descriptor of the TTY's master side back (right) to *proxy* (Step 2). *Proxy* receives the file descriptor of the TTY's master side and uses the **REXCB_NEWFD** RPC to send it to *rex* (Step 3). The arguments to **REXCB_NEWFD** are the REX channel number and current file descriptor over which the new file descriptor is begin sent, plus the new file descriptor's number. Once the *rex* client receives and processes this RPC, *rex* and *proxy* add a new REX file descriptor to the channel over which the modules can send data (Step 4). This file descriptor represents the master side of the TTY. On the

[3] Unlike traditional remote login daemons, *ptyd*, with its single system-wide daemon architecture, could easily defend against TTY-exhaustion attacks by malicious users. Currently, however, this feature is not implemented.

client machine, *rex* copies data back and forth between this copy of the TTY's master file descriptor and its own standard file descriptors (which might, for example, be connect to the *xterm* in which *rex* was started). When the user's shell running on the remote machine writes data to the TTY, *proxy* will read the data and send it over the channel's new file descriptor to *rex*.

Rex and *ttyd* also implement terminal device behavior that cannot be expressed through the Unix-domain socket abstraction. For example, typically when a user resizes an *xterm*, the application on the slave side of the pseudo-terminal receives a `SIGWINCH` signal and reads the new window size with the *ioctl* system call.

In REX, when a user resizes an *xterm* on the client machine, the program running on the remote machine needs to be notified. The *rex* client catches the `SIGWINCH` signal, reads the new terminal dimensions through an *ioctl*. It sends the new window size over the channel using file descriptor 0, which is connected to *ttyd*. Upon receiving the window resize message, *ttyd* updates the server side pseudo-terminal through an *ioctl*.

4.3.2 Forwarding X11 Connections

REX also supports X11 connection forwarding using channels and file descriptor passing. *Rex* tells *proxy* to run a module called *listen* with the argument *-x*. *Listen* finds an available X display on the server and listens for connections to that display on a Unix-domain socket in the directory `/tmp/.X11-unix`. *Listen* notifies the *rex* client of the display it is listening on by writing the display number to file descriptor 0.

Based on this remote display number, *rex* generates the appropriate `DISPLAY` environment variable that needs to be set in any X programs that are to be run. Next, *rex* generates a new (fake) `MIT-MAGIC-COOKIE-1` for X authentication. It sets that cookie on the server by having *proxy* run the *xauth* program. When an X client connects to the Unix-domain socket on the server, the *listen* program passes the accepted file descriptor over the channel to *rex*, which connects it to the local X server (i.e., it copies data between the received file descriptor and the local X server's file descriptor). *Rex* also substitutes the real cookie (belonging to the local X server) for the fake one.

4.3.3 Forwarding Arbitrary Connections

REX has a generic channel interface that allows users to connect two modules from the *rex* client command-line without adding any additional code. *Rex* creates a channel that connects the standard file descriptors of the server module program to a user-specified client module program. Unlike the TTY channels described above, here the *rex* client itself does not act as the client module. Channels, combined with file descriptor passing, allow REX users to easily build extensions such as TCP port forwarding and even SSH agent forwarding.

TCP port forwarding. Port forwarding essentially makes connections to a port on one machine appear to be connections to a different port on another machine. For example, a wireless network user concerned about eavesdropping might want to forward TCP port 8888 on his laptop securely to port 3128 of a remote machine running a web proxy. REX provides such functionality through three short utility programs: *listen*, *moduled* and *connect*. In this case, the appropriate *rex* client invocation is: `rex -m "listen 8888" "moduled connect localhost:3128" host`.

Rex spawns the *listen* program, which waits for connections to port 8888; upon receiving a connection, *listen* passes the accepted file descriptor to *rex*, which uses the **REX_NEWFD** RPC to send it over the channel to *proxy*. The *moduled* module on the server is a wrapper program that reads a file descriptor from its standard input and spawns *connect* with this received file descriptor as *connect*'s standard input and output. *Connect* connects to port 3128 on the remote machine and copies data between its standard input/output and the port. A web browser connecting to port 8888 on the client machine will effectively be connected to the web proxy listening on port 3128 of the server machine.

SSH agent forwarding. REX's file descriptor passing applies to Unix-domain sockets as well as TCP sockets. One useful example is forwarding an SSH agent during a remote login session. The *rex* client command syntax is similar to the port forwarding example, but reversed: `rex -m "moduled connect $SSH_AUTH_SOCK" "listen -u /tmp/ssh-agent-sock" host`.[4] Here, the "`-u`" flag to the *listen* module tells it to wait for connections on a Unix-domain socket called `ssh-agent-sock`. Upon receiving a connection from one of the SSH programs (e.g., *ssh*, *scp*, or *ssh-add*) *listen* passes the connection's file descriptor to the client. The *moduled/connect* combination connects the passed file descriptor to the Unix-domain socket named by the environment variable `SSH_AUTH_SOCK`, which is where the real SSH agent is listening. In the remote login session on the server, the user also needs to set `SSH_AUTH_SOCK` to be `/tmp/ssh-agent-sock`. We have written a shell-script wrapper that hides these details of setting up SSH agent forwarding.

4.3.4 Forwarding the SFS agent

When first starting up, the *sfsagent* program connects to the local SFS daemon to register itself using authenticated IPC. SFS's mechanism for authenticated, intra-machine IPC makes use of a 120-line setgid program, *suidconnect* [36]. *Suidconnect* connects to a protected, named Unix-domain socket, sends the user's credentials to the listening process, and then passes the connection back to the invoking program.[5] Though *suidconnect* predates REX, REX's file descriptor passing was sufficient to implement SFS agent forwarding with no extra code on the server. Simply running *suidconnect* in a REX channel causes the necessary file descriptor to be passed back over the network to the agent on a different machine.

Once the *sfsagent* is available on the remote machine, the user can access it using RPC. All of the user's configuration is stored in one place; requests are always forwarded back to the agent, so the user does not see different behavior on different machines.

4.3.5 Same Environment Everywhere

When a user logs into a remote machine, he should have the same environment as on the local machine, even if the remote machine is in a different administrative domain. Providing the same environment across domains is particularly important because the remote machines involved often have different configurations than machines in the local domain. For example, the remote administrator might not have set up the same trust relationships or have mounted the same network file systems that are typically available in the user's local domain.

REX provides a consistent computing environment during remote login by forwarding the *sfsagent*, which stores various state for the user. For example, the agent maintains a per-user view of the `/sfs` directory (where all remote SFS file systems are mounted). When the user initiates a remote login, the combination of REX, SFS, and the agent replicates the local `/sfs` on the remote machine.

The agent is responsible for other state related to the user's computing environment as well: server key management, user authentication, revocation. Thus, the remote login session behaves

[4] When possible, *listen* rejects Unix-domain connections from other user IDs (through permission bits, `getpeereid`, or SO_PEERCRED `ioctls`). As this doesn't work for all operating systems, in practice we hide forwarded agent sockets in protected subdirectories of `/tmp/`.

[5] `getpeereid`, when available, is used to double-check *suidconnect*'s claimed credentials.

the same as the local one. For example, not only does the user have access, through the global file system, to his home directory, but he can also authenticate to it in the same way. SSH differs from this architecture in that an SSH user's environment might depend on the contents of his `.ssh` directory, which might be different between the local and remote machines. Because SSH cannot forward the *sfsagent*, even an SSH user whose home directory is on SFS would be unable to access it without starting an agent on the remote machine, which is inconvenient when using many machines.

4.4 Security

The REX architecture provides three main security benefits. First, REX minimizes the code that a remote attacker can exploit. Second, REX allows users to configure and manage trust policies during a remote login session. Third, REX provides several ways to name remote servers that are designed to avoid man-in-the-middle attacks.

4.4.1 Minimizing Exploitable Code

Remote exploits are a major concern for software developers. Buffer overruns and other bugs have led to serious system security compromises. Protecting against remote exploits is critical, especially when the remote execution service is open to logins from other administrative domains. In such configurations incoming connections are not restricted to trusted, local users, but can be initiated by any Internet user.

REX attempts to mitigate this problem by minimizing the amount of remotely exploitable code. REX also attempts to protect against local exploits by minimizing the amount of code that runs with superuser privileges. REX offers protection against both types of exploits through the REX architecture's use of local file descriptor passing.

In REX, only *rexd* listens for and accepts connections from remote clients. *Rexd* runs with superuser privileges in order to authenticate the user (via *sfsauthd*) and then spawn *proxy* with the privileges of that user. *Rexd* uses local file descriptor passing to pass the client connection to *proxy*.

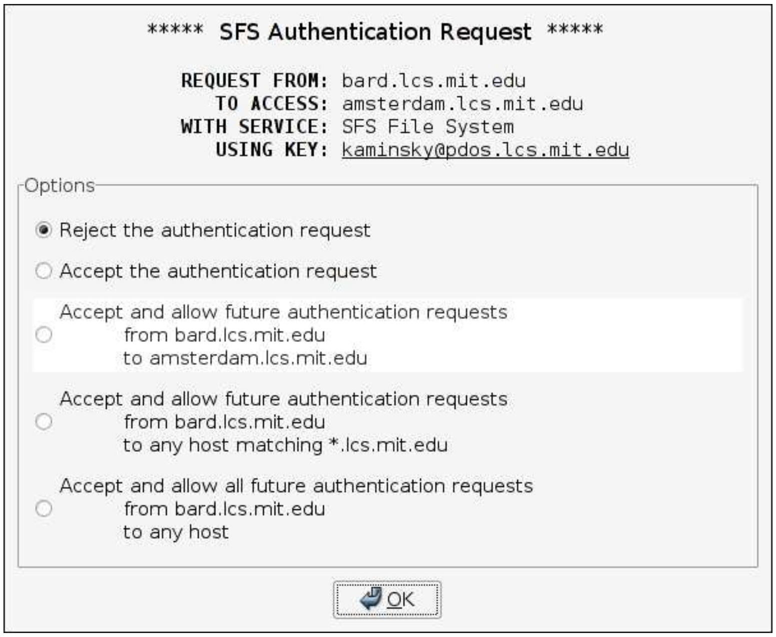
REX reduces the potential of local superuser exploits. For example, the privileged *ptyd* daemon allocates pseudo-terminals and passes them, using local file descriptor passing, to *ttyd* which runs with the privileges of a normal user. The privileged programs are small and perform only a single task, allowing easy auditing. Not counting general-purpose RPC and crypto libraries from SFS, *rexd* is about 500 lines of code and *ptyd* is about 400 lines.

Protecting against exploits in REX is straightforward because the architecture and protocols were designed from the ground up to allow for easy privilege separation. Thinking about privilege separation and other security issues is an important principle in designing new protocols.

4.4.2 Restricted Delegation of Authority

One particularly difficult issue with remote login is the problem of accurately reflecting users' trust in the various machines they log into. Typically, users trust machines in their own administrative domain, but they may trust machines in other domains less.

For example, a user may use local machine *A* to log into remote machine *B*, which is part of a remotely administered shared computing cluster. From *B*, the user logs back into *A*. Many utilities support credential forwarding to allow password-free login from *B* back to *A*—but the user may not trust machine *B* as much as machine *A*. For this reason, other systems often disable credential forwarding by default, but the result of that is even worse. Users logging from *B* back into *A* will



***** SFS Authentication Request *****

```
REQUEST FROM: bard.lcs.mit.edu
   TO ACCESS: amsterdam.lcs.mit.edu
WITH SERVICE: SFS File System
   USING KEY: kaminsky@pdos.lcs.mit.edu
```

Options

● Reject the authentication request

○ Accept the authentication request

○ Accept and allow future authentication requests
from bard.lcs.mit.edu
to amsterdam.lcs.mit.edu

○ Accept and allow future authentication requests
from bard.lcs.mit.edu
to any host matching *.lcs.mit.edu

○ Accept and allow all future authentication requests
from bard.lcs.mit.edu
to any host

Figure 4-8: A GUI confirmation program

simply type their passwords. This approach is both less convenient and less secure, as untrusted machine *B* will now not only be able to log into *A*, it will learn the user's password!

To address this dilemma, REX and the *sfsagent* support restricted delegation of authority through a mechanism called *selective signing*. Selective signing offers a convenient way to build up trust policies incrementally without sacrificing security. During remote login, REX remembers the machines to which it has forwarded the agent. In the remote login session, when the user invokes *rex* again and needs to authenticate to another server, his *sfsagent* will run a user-specified *confirmation program*. This program, which could be a simple text message or a graphical pop-up dialog box, displays the name of the machine originating the authentication request, the machine to which the user is trying to authenticate, the service being requested (e.g., REX or file system) and the key with which the agent is about to sign. The user's agent knows about all active REX sessions and forwarded agent connections, so the remote machine cannot lie about its own identity.

Because signed authentication requests contain the name and public key of the server being accessed, as well as the particular service, the agent always knows exactly what it is authorizing. With this information, the user can choose whether or not to sign the request. Thus, users can decide case-by-case whether to let their agents sign requests from a particular machine, depending on the degree to which they trust that machine. The modularity of the agent architecture allows users to plug in arbitrary confirmation programs. Currently, SFS comes with a GUI program (see Figure 4-8) that displays the current authentication request and the key with which the agent is about to sign it. The user has five options: to reject the request; to accept (sign) it; to sign it and automatically sign all similar requests in the future; to sign it and all similar requests where the server being accessed is in the same DNS domain as the given server; and to sign it and all subsequent requests from the same client, regardless of the server being accessed.

4.4.3 Naming Servers Securely

Securely connecting to a remote machine requires securely naming and authenticating that machine. At the lowest level, REX uses self-certifying hostnames to create such connections. The system also provides, based on the server authentication mechanisms available in SFS, several ways to retrieve these self-certifying hostnames automatically. These mechanisms free the user from having to type self-certifying hostnames directly (though, of course, they are free to do so if desired). Having a variety of secure, flexible server naming techniques is important in a cross-domain environment because different users have different needs, which can depend on the particular machine they are accessing.

The primary technique that REX uses to retrieve self-certifying hostnames is the Secure Remote Password (SRP) protocol [62]. Using SRP, the user can authenticate the server based on a weak password. The user registers for SRP ahead of time (e.g., by visiting a system administrator or perhaps by bootstrapping with a Unix password). When the user runs `"rex host"`, the REX client software will prompt the user for his SRP password and execute the protocol. If successful, the user will obtain a copy of the server's self-certifying hostname, which REX will use to create the secure remote login connection. The *sfsagent* maintains a cache that maps DNS names (names that the user types in) to their self-certifying hostnames. On successful completion of the SRP protocol, REX adds a new mapping to the agent's cache. Later, if the user decides to connect to the same server, *rex* can ask the agent for the server's self-certifying hostname without having to use SRP (and therefore prompt the user for a password).

A second technique for securely naming servers is to use symbolic links in the file system. Self-certifying hostnames allow the file system to double as a key management namespace. For example, `/sfs/mit` could be a symbolic link pointing to self-certifying hostname of an MIT login server. Users could REX directly to such a link. If the owner of the link updates it, that change is reflected immediately the next time a user accesses it.

Finally, advanced users can employ *dynamic server authentication*. Here, users install *certification programs* into their agents, which translate human-readable names into self-certifying names on-the-fly. Dynamic server authentication is discussed more fully in SFS papers [38, 27].

Chapter 5

REX Evaluation

We have implemented REX as part of the SFS computing environment. This chapter quantifies REX's extensible architecture in terms of code size. This comparison shows that the privileged code in REX is small compared to the unprivileged code, which contains most of REX's features and extensions. The chapter then compares the performance of REX with the OpenSSH [43] implementation of SSH protocol version 2 [64]. The measurements demonstrate that the extensibility gained from file descriptor passing comes at little or no cost.

5.1 Code Size

REX has a simple and extensible design; Table 5.1 lists the (approximate) relative code sizes of different pieces of the system. REX's wire protocol specification is only 250 lines of Sun XDR code [54]. REX has two component programs that run with enhanced privileges (shown in bold). *Rexd* receives incoming REX connections and adds only 500 lines of trusted code to the system (not counting the general-purpose RPC and crypto libraries from the SFS toolkit [37]). *Ptyd* allocates pseudo-terminals to users who have successfully authenticated and is about 400 lines of code.

Proxy runs with the privileges of the authenticated users and is just over 1000 lines of code; the *rex* client is about 2,350 lines. Extensions to the *sfsagent* for connection caching consist of less than 900 lines of code.

Program	Lines of Code
XDR Protocol	250
rexd	500
ptyd	400
proxy	1000
rex+agent	3250
listen	250
moduled	30
connect	375
ttyd	260

Table 5.1: REX code size comparison. The numbers in this table are approximate and do not include general-purpose RPC and crypto libraries from SFS. Programs shown in bold run with superuser privileges.

Protocol	Average Latency (msec)	Minimum Latency (msec)
SSH	121	120
REX (initial login)	51	50
REX (subsequent logins)	21	20

Table 5.2: Latency of SSH and REX logins

Modules that extend REX's functionality are also small. The *listen*, *moduled*, and *connect* modules are approximately 250, 30, and 375 lines of code, respectively. *Ttyd* is under 260 lines.

If REX were to gain a sizable user base, we could expect the code size to grow because of demands for features and interoperability. The code growth, however, would take place in untrusted components such as *proxy* or in new external modules (likely also untrusted). Because of the extensibility, well-defined interfaces, and the use of file descriptor passing, the trusted components can remain small and manageable.

5.2 Performance

We measured the performance of REX and OpenSSH 3.8p1 [43] on two machines running Debian with a Linux 2.4 kernel. The client machine consisted of a 2 GHz Pentium 4 with 512 MB of RAM. The server machine consisted of a 1.1 GHz AMD Athlon with 768 MB of RAM. A 100 Mbit, switched Ethernet with a 118 μsec round-trip time connected the client and server. Each machine had a 100 Mbit Ethernet card.

We configured REX and SSH to use cryptographic systems of similar performance. For authentication and forward secrecy, SFS was configured to use the Rabin-Williams cryptosystem [60] with 1,024-bit keys. SSH uses RSA with 1,024-bit keys for authentication and Diffie-Hellman with 768-bit ephemeral keys for forward secrecy. We configured SSH and SFS to use the ARC4 [30] cipher for confidentiality. For integrity, SFS uses a SHA-1-based message authentication code while SSH uses HMAC-SHA-1 [16, 33]. Our SSH server had the privilege separation feature [46] enabled.

5.2.1 Remote Login

We compare the performance of establishing a remote login session using REX and SSH. We expect both SSH and REX to perform similarly, except that REX should have a lower latency for subsequent logins because of connection caching.

Table 5.2 reports the average and minimum latencies of 100 remote logins in wall clock time. In each experiment, we log in, run `/bin/true`, and then immediately log out. The user's home directory is on a local file system. For both REX and SSH, we disable agent forwarding, pseudo-tty allocation, and X forwarding.

The results demonstrate that an initial REX login is faster than an SSH login. In both cases, much of the time is attributable to the computational cost of modular exponentiations. An initial REX connection requires two concurrent 1,024-bit Rabin decryptions—one by the client for forward secrecy, one by the server to authenticate itself—followed by a 1,024-bit Rabin signature on the client to authenticate the user. All three operations use the Chinese Remainder Theorem to speed up modular exponentiation.

An SSH login performs a 768-bit Diffie-Hellman key exchange—requiring two 768-bit modular exponentiations by each party—followed by a 1,024-bit RSA signature for server authentication and

Protocol	Throughput (Mbit/sec)	Round-Trip Latency (μsec)
TCP	87.1	118
SSH	86.2	294
REX	86.0	394

Table 5.3: Throughput and latency of TCP port forwarding

a 1,024-bit RSA signature for user authentication. The Diffie-Hellman exponentiations cannot be Chinese Remaindered, and thus are each more than 50% slower than a 1,024-bit Rabin decryption. The RSA operations cost approximately the same as Rabin operations.

The cost of public key operations has no bearing on subsequent logins to the same REX server, as connection caching requires only symmetric cryptography. Development versions of OpenSSH have started to implement a connection caching-like ability [40]; with this feature, we would expect performance similar to REX's on subsequent logins.

5.2.2 Port Forwarding Throughput and Latency

Both SSH and REX can forward ports and X11 connections. To demonstrate that REX performs just as well as SSH, we measure the throughput and round-trip latency of a forwarded TCP port with NetPipe [52]. NetPipe streams data using a variety of block sizes to find peak throughput and minimum round-trip latency. The round-trip latency represents the time to send one byte of data from the client to the server, and receive acknowledgment.

Table 5.3 shows the maximum throughput and minimum round-trip latency of three different types of connections. First, we measure the throughput and latency of an ordinary, insecure TCP connection. Next, we measure the throughput and latency of a forwarded port over established SSH and REX connections. The results show that file descriptor passing in REX does not noticeably reduce throughput.

We attribute the additional latency of ports forwarded through REX to the fact that data must be propagated through both *proxy* and *connect* on the server, incurring an extra context switch in each direction. If *rex* and *proxy* provided a way to "fuse" two file descriptors, we could eliminate the inefficiency. Note, however, that over anything but a local area network, actual propagation time would dwarf the cost of these context switches.

Chapter 6

Related Work

The user authentication and remote execution systems described in this thesis are designed to operate effectively across administrative domains. The authentication server provides an important property—cross domain groups—that few other systems provide in practice. Its main contribution relative to these systems is a simple design and implementation that is practical and works well in a file system context. REX builds upon the success of SSH, but provides a new architecture based on file descriptor passing that allows for easy extensions without compromising security. This chapter describes the relevant previous systems. Both systems draw on ideas from and address limitations in previous systems.

6.1 User Authentication

As mentioned in Chapter 1, there are certain desirable properties that one might want in a user authentication system: flexibility, bounded staleness, scalability, simplicity, and privacy. The authentication server described in this thesis offers a new design point with respect to these properties. It provides the flexibility to name remote users and groups and a simple usage model that mirrors local access control mechanisms. It has limited freshness, but can bound staleness; has scalability reasonable for a file system context; and provides limited privacy for group lists. This section discusses several previous approaches to user authentication and how they relate to the authentication service presented above.

6.1.1 Kerberos/AFS

The AFS [25] file system, combined with the Kerberos authentication system [55], provides a secure distributed file system with centralized user authentication, but it does not support cross-domain groups. Kerberos principals are organized into realms which are usually defined along administrative boundaries. AFS has no mechanism to refer to remote principals; users can place only 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.

AFS makes different tradeoffs than our authentication server. AFS does not have the flexibility to reference remote groups (or even nested groups in general). AFS has reasonable freshness; staleness is bounded because Kerberos tickets have a fixed lifetime (assuming roughly synchronized clocks). Like the authentication server, AFS has a simple, familiar usage model based on users, groups, and ACLs. Because AFS does not support remote groups, privacy of group lists and scalability are not applicable.

6.1.2 Microsoft Windows 2000

Microsoft Windows 2000 servers [39] differ from the authentication server and SFS primarily in that they require an organized trust infrastructure. The Windows system supports 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.

The Windows 2000 domain system 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.

By requiring an organized trust infrastructure, the Windows domain system limits flexibility by preventing users from naming arbitrary remote users and groups in their group membership lists without first having their administrators set up trust relationships. In choosing to have multiple group types, the Windows domain system sacrifices simplicity, not only in terms of their design (and presumably their implementation) but also in terms of their usage model. 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). Windows domain administrators can configure the frequency with which group information is replicated, providing bounded staleness similar to the authentication server.

6.1.3 Certificate-Based Systems

Certificate-based authentication systems can provide the flexibility to name remote users and groups, scalability to millions of users per group (e.g., naming all of the Visa users in a country), and privacy, but they often lack simplicity and freshness. For example, instead of a familiar usage model based on users, groups and ACLs, users must often deal directly with certificates, certificate chains, and proofs of group membership; furthermore, many of these systems require that the user (client) discover and present these identity proofs to the server. In SFS, the authentication server has the burden of determining the groups to which the user belongs. Assigning this burden to the client, however, enables certificate-based systems to provide better scalability and privacy. Finally, certificate-based systems often rely on trusted third-party certificate authorities (CAs) that are offline; consequently, freshness suffers because certificate expiration times are long (e.g., a year). This section describes two popular certificate-based systems in more detail. Several other systems [4, 26, 9, 34] are based on this model.

The Taos operating system [35, 61] and the Echo file system [7] 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 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. [21] and Birrell et al. [8] discuss similar hierarchic authority trees and suggest “symbolic links” or “cross-links” as a way to allow one CA 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 CAs to give meaning to those names.

SPKI/SDSI [15, 48] 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 [63], and web-of-trust based systems such as PGP [66]. SPKI/SDSI principals are essentially public keys, which define a local name space. Users who 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 [11]. 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.

6.1.4 Exposing Public Keys

The authentication server takes the idea of exposing public keys to users via self-certifying hostnames and applies it to user authentication. The idea of exposing public keys has its basis in earlier systems such as PGP [66]. SSH [65] also allows users to directly manipulate public keys in the context of remote login. Users can place a copy of their public keys in an `authorized_keys` file on the server. This `authorized_keys` file acts like an ACL for remote login.

In the context of file systems, Farsite [3] allows users to place public keys on ACLs. Farsite, however, does not cross administrative domains; its target is a large corporation or university. Other file systems [47, 41] expose keys by providing capabilities. These systems (like SFS) allow users who do not have a local account to access the file system, but they do not provide a traditional file sharing interface based on users, groups, and ACLs (instead, users must present the appropriate capability).

6.2 Remote Execution

Several tools exist for secure remote login and execution. This section focuses primarily on those tools but concludes with a discussion of agents and file descriptor passing.

6.2.1 SSH

SSH [65] is the de-facto standard for secure remote execution and login. REX differs from SSH primarily in its architecture. REX uses file descriptor passing plus external modules to implement many of the features available in both system. In REX, these features are built outside of the core REX software, and they do often do not require any protocol changes, or even recompilation. In SSH, they are typically part of the main code base.

The REX architecture also attempts to limit exploits by placing code that handles incoming network connections and code that runs with superuser privileges in separate processes. The latest versions of OpenSSH [43] have also embraced privilege separation [46, 50], but the SSH protocol was not designed to facilitate such an architecture. The complexity of the implementation reflects this fact. For example, in one step, SSH must extract the memory heap from a process and essentially recreate it in another process's address space. Moreover, even the least privileged, "jailed," SSH processes still require the potentially dangerous ability to sign with the server's secret key.

Aside from file descriptor passing and integration with SFS, REX offers several features not presently available in SSH. REX's connection caching improves connection latency. Connection resumption and NAT support (from the underlying SFS infrastructure) allow REX to operate transparently over a wider variety of network configurations. Selective signing improves security in mixed-trust environments and saves users from typing their passwords unnecessarily. Conversely, SSH provides features not present in REX, notably compatibility with other user-authentication standards.

6.2.2 Other Secure Remote Login Utilities

Before SSH, researchers explored other options for secure remote login [32, 57]. Kim et al. [32] implemented a secure *rlogin* environment using a security layer beneath TCP. The system defended against vulnerabilities created by hostname-based authentication and source address spoofing. Secure *rlogin* used a modular approach to provide a flexible security policy. Like REX, secure *rlogin* used small, well-defined module interfaces. REX uses a secure TCP-based RPC layer implemented by SFS; secure *rlogin* used a secure network layer between TCP and IP, similarly to IPSec [31].

Kerberized remote login is based on a centralized architecture, and therefore requires a trusted third party for client-server authentication. REX and SFS both support third-party authentication, but do not require it, and in practice they are often used without it. Because AFS uses Kerberos for authentication, Kerberized remote login can authenticate users to the file system before logging them in. REX provides similar support for the SFS file system.

The Globus [17] Project provides a Grid metacomputing infrastructure that supports remote execution and job submission through a resource allocation manager called GRAM [13] and access to global storage resources through GASS [6]. Globus was designed to provide a uniform interface to distributed, remote resources, so individual client users do not need to know the specific mechanisms that local resource managers employ. By default, GRAM and GASS provide simple output redirection to a local terminal for programs running on a remote machine. Tools built on top of Globus can offer features such as pseudo-terminals, X11 forwarding and TCP port forwarding [22].

These features, however, seem to be built into the software and protocol whereas REX provides the same extensibility and security (privilege separation) through file descriptor passing.

6.2.3 Agents

While REX is not the first remote execution tool to employ user agents, it makes far more extensive use of its agent than other systems. The SFS agent holds all of the user's state and provides access to it through an RPC interface. Encapsulating all state behind an RPC agent interface allows a user's configuration to be propagated from machine to machine simply by forwarding an RPC connection. By contrast, the SSH agent is responsible for essentially just authenticating users to servers. For other tasks such as server authentication, however, SSH relies on configuration files (e.g., `known_hosts`) in users' home directories. When users have different home directories on different machines, they see inconsistent behavior for the same command, depending on where it is run.

Another significant difference between the REX and SSH agents is that the SSH agent returns authentication requests that are not cryptographically bound to the identity of the server to which they are authorizing access. As a result, a remote SSH client could lie to the local agent about what server it is trying to log into. Concurrently and independently of REX, the SSH agent added support for a simple confirmation dialog feature, but the SSH agent is unable to build up any meaningful policies or even tell the user exactly what is being authorized.

Other systems do, however, provide restricted credential forwarding. For example, global identities within the Grid Security Infrastructure (GSI) [9, 18], part of the Globus project, are based on X.509 [63] public-key certificates and the SSL/TLS protocols [14]. Recent extensions to GSI add support for proxy certificates [59], which allow an entity to delegate an arbitrary subset of its privileges. A new GSI-enabled version of SSH can use these proxy certificates to provide limited delegation to applications running on the remote machine, similarly to REX's selective signing mechanism.

Finally, the security architecture for the Plan 9 system [12] has an agent, *factotum*, which is similar to the SSH and SFS agents. *Factotum*, however, is implemented as a file server.

6.2.4 File Descriptor Passing

Though local Unix file descriptor passing is used in a number of systems, REX extends this idea to work between processes running on different machines. OpenSSH, for example, as part of its privilege separation code, internally uses local file descriptor passing to handle pseudo-terminals. Though file descriptor passing is part of the source code, it is not part of the protocol. Generalizing the idea cleanly to pass file descriptors for other purposes would require modification to the SSH protocol, which we hope people will consider in future revisions.

An alternative to file descriptor passing is file namespace passing, as is done in Plan 9 [44]. Plan 9's *cpu* command can replicate parts of the file namespace of one machine on another. When combined with device file systems like `/dev/fd`, this mechanism effectively subsumes file descriptor passing. Moreover, because so much of Plan 9's functionality (including the windowing system) is implemented as a file system, *cpu* allows most types of remote resource to be accessed transparently. Unfortunately, Unix device and file system semantics are not amenable to such an approach, which is one of the reasons tools like SSH have developed so many different, ad hoc mechanisms for handling different types of resources.

Chapter 7

Conclusion

The Internet has enabled a new kind of collaboration in which users access and share resources between different administrative domains. In a single domain, users have convenient, easy tools for sharing both data and machines: a network file system and a remote execution utility. Extending these tools to work across administrative boundaries presents a number of challenges, particularly with respect to security.

This thesis presents a new user authentication service and a new remote execution utility designed to address these challenges. Section 7.1 summarizes their contributions, designs, and implementations. As usual with research, answering one question results in more questions. Section 7.2 identifies some open problems.

7.1 Summary

The goal of the authentication server and REX is to make using and sharing resources across administrative domains just as easy and secure as using and sharing resources with a single domain. The authentication service contributes a new design point in the space of user authentication. It provides cross-domain groups for access control and a familiar, intuitive interface for sharing resources. Users create groups, place local and remote principals in those groups, and then place those new groups on their ACLs. A main goal is simplicity of design and implementation. Thus, the system restricts ACLs to contain only local users and groups, allowing for a separation of the service into two components: an authentication server and the file server. The authentication server pre-fetches and caches relevant remote authentication information periodically in the background; at login time, it uses that information to issue credentials (e.g., group membership) for the user. The file server uses these credentials and ACLs to make access control decisions.

The authentication service design makes several tradeoffs. To maintain file system availability, the system does not provide perfect freshness. Local authentication servers can, however, bound the staleness of the information that they use. Experiments demonstrate that the server can scale to groups with tens of thousands of users, reasonable for a file system context. The system's server-pull design has limited privacy with respect to group lists. Finally, the ability to name remote users and particularly remote groups means that the system has all-or-nothing delegation.

REX provides secure remote login and execution in the tradition of SSH, but it is designed from the ground-up with a new architecture centered around extensibility and security, two important properties in cross-domain settings. REX's extensibility, based on emulated file descriptor passing between machines, allows users to add new functions to REX without changing the protocol. REX and the SFS agent also provide users a consistent environment across machines. REX's security benefits are a limited amount of exploitable code and a convenient mechanism for limited delegation of authority when accessing less-trusted machines.

Both the authentication server and REX are implemented as part of the SFS computing environment. The SFS distribution is open-source and available at `http://www.fs.net/`.

7.2 Open Problems

The section describes several open problems related to the user authentication service and some extensions and/or improvements to addresses these questions.

7.2.1 Changing Server Keys

In the current design of the authentication system, changing or revoking an authentication server's public-private key pair is inconvenient. Hashes of the public keys appear in the self-certifying hostnames that are part of remote user and group names. These user and group names can appear in a number of different groups across several authentication servers. Changing the keys involves updating all of those group records with new self-certifying user and group names.

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. Using this approach, authentication servers could allow individuals to name remote users and group (i.e., 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 [38]. 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.

7.2.2 Usability

Ease-of-use is an important issue in deploying secure systems that deal directly with keys and hashes. For example *sfskey* currently requires users to specify self-certifying hostnames explicitly when modifying a group list. Most users, however, do not want to type self-certifying hostnames. For convenience, *sfskey* could allow users to specify a symbolic link in the file system instead. *Sfskey* would dereference this link and store the resulting self-certifying hostname in the group list. Other server naming techniques from SFS might also be useful. For example, *sfskey* could look up DNS names using SRP (including the agent's cache) or using dynamic server authentication. In general, *sfskey* could use the secure server naming techniques described in Section 4.4.3.

7.2.3 Site Policies

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, such as academia or free-software development communities, this flexibility is welcome. In some environments, such as 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.

7.2.4 Reducing Server Load

The load on remote authentication servers is an important design consideration, particularly for servers that provide large and/or popular groups. The current fetching strategy requires local authentication servers always to contact the authoritative remote server that defines a given remote user or group group. If many references to a particular remote server exist, that server might have to handle many incoming network connections and send back gigabytes of data.

To reduce load on remote authentication servers, local servers could fetch group records from replicas. These replicas, however, should not need to know the authoritative server's private key. One way to achieve this goal might be to have the authoritative authentication server sign its user and group records. Then, local servers could retrieve these signed records from untrusted replicas.

A second way to use replicas to reduce server load might be to employ a different technique to compute the deltas between the locally cached version of a group and the authoritative version. Currently, the remote authentication server stores change logs for each group, which it uses to transfer the deltas. An alternative would be to use an "rsync"-style [2, 56] approach in which only the differences between the remote and local copies of a group are sent. With this strategy, a local authentication server could retrieve a copy of the group from an untrusted replica (possibly even a nearby, higher-bandwidth authentication server's cache). Even though this copy might be out-of-date or incorrect, the local server could then quickly check it against the authoritative version (on the authoritative authentication server) and retrieve the differences.

7.2.5 Privacy

Inherent in the design of the authentication service is that group membership lists are public. Section 2.7 describes several possible techniques to provide some degree of privacy within the current system. These techniques, however, at best only obfuscate private information behind public key hashes or arbitrary identifiers.

A tradeoff exists between the privacy that a system provides and the burden placed on a user when accessing a resource. One end of the spectrum is an authentication system that provides complete privacy—all group membership lists are private and only the owners of the group know its membership. Even the server providing the resource does not know the entire group membership, only that the user currently accessing the resource is a member of a particular group on the resource's ACL. The server also does not know about any other groups to which the user belongs, except the one on its ACL. The tradeoff in such a system is that the user must know exactly how he merits access to the resource. That is, he must know at least one group that is on the ACL and prove to the server that he is a member of that group. Client-supplied certificate chains are an example of such a system.

On the other end of the spectrum, the system could provide little or no privacy by having public group lists. Here, the server already knows the members of every group on all of its resource's ACLs. The burden on the user, however, is minimal. He simply proves his identity to the server (e.g., that he possesses some public-private key pair), and the server checks to see if the user merits access. The authentication system described in this thesis is an example of this privacy-burden tradeoff.

Other design points exist with respect to this tradeoff. For example, a system could still have private group lists but reduce the burden on the user by allowing the server to know all of the user's memberships. In a certificate-based system, the user could provide certificate chains for all groups of which he is a member. The server would use the proof that is relevant for the resource/ACL the user is trying to access. Here, the user does not need to know anything about the groups on the ACL ahead of time, but the server will know more than strictly necessary about the user.

Other privacy tradeoffs are also possible. For instance, including ACLs on user and group records could improve privacy in a system like the authentication server where these records are shared publicly. Then, only specific authentication servers could fetch certain private group lists. This design decision would increase complexity, however. First, record owners would have to manage these new ACLs, which name the servers (or users) that can fetch the record. Second, nested groups might require further policies. If server *B* has permission to fetch a group record on server *A*, does server *C*, which includes a group on *B* have permission to fetch the record from *A* also? If so, to how many levels of nesting does that permission extend?

Bibliography

[1] *Concurrent Versions System*. URL `http://www.gnu.org/manual/cvs/index.html`.

[2] *rsync*. URL `http://samba.anu.edu.au/rsync/`.

[3] 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.

[4] 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.

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

[6] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, May 1999.

[7] 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.

[8] 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, April 1986.

[9] Randy Butler, Douglas Engert, Ian Foster, Carl Kesselman, Steven Tuecke, John Volmer, and Von Welch. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

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

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

[12] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, San Francisco, CA, August 2002.

[13] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, Orlando, Florida, March 1998.

[14] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246, Network Working Group, January 1999.

[15] 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.

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

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

[18] Ian Foster, Carl Kesselman, G. Tsudik, and Steven 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.

[19] 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.

[20] fsh — Fast remote command execution. `http://www.lysator.liu.se/fsh/`.

[21] 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.

[22] glogin. `http://www.gup.uni-linz.ac.at/glogin/`.

[23] Andreas Grünbacher. POSIX access control lists on Linux. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 259–272, San Antonio, TX, June 2003.

[24] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Network Working Group, February 2000.

[25] 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.

[26] 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.

[27] Michael Kaminsky. Flexible key management with SFS agents. Master's thesis, Massachusetts Institute of Technology, May 2000.

[28] Michael Kaminsky, Eric Peterson, Daniel B. Giffin, Kevin Fu, David Mazières, and M. Frans Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 199–212, Boston, Massachusetts, June 2004.

[29] Michael Kaminsky, George Savvides, David Mazières, and M. Frans Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 60–73, Bolton Landing, New York, October 2003.

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

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

[32] Gene Kim, Hilarie Orman, and Sean O'Malley. Implementing a secure rlogin environment: A case study of using a secure network layer protocol. In *Proceedings of the 5th USENIX Security Symposium*, pages 65–74, Salt Lake City, UT, June 1995.

[33] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, February 1997.

[34] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, 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, Cambridge, MA, November 2000.

[35] 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.

[36] David Mazières. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology, May 2000.

[37] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*, pages 261–274, Boston, Massachusetts, June 2001.

[38] 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, Kiawa Island, SC, December 1999.

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

[40] Damien Miller. `http://www.mindrot.org/pipermail/openssh-unix-dev/2004-June/021522.html`, June 16, 2004, openssh-unix-dev@mindrot.org.

[41] 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.

[42] P. Mockapetris. Domain Names—Concepts and Facilities. RFC 1034, Network Working Group, November 1987.

[43] OpenSSH. `http://www.openssh.com/`.

[44] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *ACM SIGOPS Operating System Review*, 27(2):72–76, April 1993.

[45] Dave Presotto and Dennis Ritchie. Interprocess communication in the Eighth Edition Unix system. In *Proceedings of the 1985 Summer USENIX Conference*, pages 309–316, Portland, OR, June 1985.

[46] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, Washington, DC, August 2003.

[47] 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, DC, August 2001.

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

[49] 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.

[50] Jerome Saltzer. Protection and control of information in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[51] George Savvides. Access control lists for the self-certifying filesystem. Master's thesis, Massachusetts Institute of Technology, August 2002.

[52] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996. `http://www.scl.ameslab.gov/netpipe`.

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

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

[55] 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.

[56] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1997.

[57] David Vincenzetti, Stefano Taino, and Fabio Bolognesi. Stel: Secure telnet. In *Proceedings of the 5th USENIX Security Symposium*, pages 75–84, Salt Lake City, UT, June 1995.

[58] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol, (v3). RFC 2251, Network Working Group, December 1997.

[59] Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Steven Tuecke, Jarek Gawor, Sam Meder, and Frank Siebenlist. X.509 proxy certificates for dynamic delegation. In *Proceedings of the 3rd Annual PKI R&D Workshop*, Gaithersburg, MD, April 2004.

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

[61] 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.

[62] 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.

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

[64] T. Ylönen and D. Moffat (Ed.). SSH Transport Layer Protocol. Internet draft (draft-ietf-secsh-transport-17.txt), Network Working Group, October 2003. Work in progress.

[65] 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.

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