REX: Secure, Extensible Remote Execution

Michael Kaminsky, Eric Peterson, Daniel B. Giffin,
Kevin Fu, David Mazières, M. Frans Kaashoek

MIT Computer Science and Artificial Intelligence Laboratory,
NYU Department of Computer Science

kaminsky@csail.mit.edu, ericp@csail.mit.edu, dbg@cs.nyu.edu,
fubob@csail.mit.edu, dm@cs.nyu.edu, kaashoek@csail.mit.edu

Abstract

The ubiquitous SSH package has demonstrated the importance of secure remote login and execution. As remote execution tools grow in popularity, users require new features and extensions, which are difficult to add to existing systems. REX is a remote execution utility with a novel architecture specifically designed for extensibility as well as security and transparent connection persistence in the face of network complexities such as NAT and dynamic IP addresses. 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. Finally, REX provides mechanisms for accessing servers without globally routable IP addresses, and for resuming sessions when a TCP connection aborts or an endpoint's IP address changes. Measurements of the system demonstrate that REX's architecture does not come at the cost of performance.

1 Introduction

Remote login and execution are network facilities that many people need for their day-to-day computing. 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 tools have become quite complex.

The popular SSH [38] program demonstrates that users expect 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. Many of these features require changes to the remote login protocol, for which developers add new message types.

Moreover, many users want other features that are not yet available: limitations on the amount of code subject to remote exploits, convenient trust management policies, transparent access to servers behind network address translation (NAT), and support for long-running remote login sessions when the client and server both change their IP addresses. The challenge in designing and building a remote execution tool is to address this diverse set of needs in a single, simple, extensible framework.

This paper introduces a new remote login and execution utility called REX, which has three main goals: extensibility, security, and transparent connection persistence despite NAT and dynamic IP addresses. 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.

Extensibility. REX's approach to extensibility is for the core system and protocol to provide the simplest possible interface on which external utilities can implement advanced features like remote pseudo-terminal access, port forwarding, and authentication delegation. This interface consists principally of file descriptor passing: a process on one machine can 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 authentication agent connection, or some as-yet-unanticipated future extension.

Security. REX was designed from the ground up to minimize 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. This separation of functions and privileges is possible because *rexd* uses local file descriptor passing to hand off incoming connections to *proxy*.

The latest versions of OpenSSH [21] have moved in a similar direction by embracing privilege separation [24, 26]. The SSH protocol, however, was not designed to facilitate such an architecture, and 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.

A second security goal of REX is to provide precise delegation of authority to processes running on remote machines. Delegation of authority allows a remote process to access and authenticate itself to remote resources. However, a user might trust the remote machine less than the local one. To address this problem, REX can prompt users to authorize remote accesses on a case-by-case basis; by optionally instructing the agent to allow similar accesses in the future, users can build trust policies incrementally.

Transparent Connection Persistence. REX provides transparent connection persistence in the face of complex network configurations. The prevalence of network address translation (NAT) makes it hard to run globally accessible servers on many machines, while dynamically assigned IP addresses can disrupt long-running sessions. REX lets users transparently connect to remote login servers behind NAT boxes using either an externally addressable proxy server or DNS SRV records [12] (in conjunction with static TCP port mapping). REX's automatic connection resumption allows clients and servers to change IP addresses during the course of a connection.

We have built REX as part of the SFS [20] computing environment. REX currently offers modules that handle pseudo-terminal support, TCP port forwarding, X11 forwarding with cookie authentication, and Unix-domain socket forwarding. REX adds only two small pieces of privileged code to the system. One of these, the pseudo-terminal daemon, is only 400 lines of code and never touches an Internet socket; it is therefore unlikely to be remotely exploitable. The other, *rexd*, is only 500 lines of code (not counting general-purpose crypto and Remote Procedure Call libraries). REX is in daily use, it runs on Unix, and the source code is freely available.

The rest of the paper is structured as follows. Section 2 describes REX's architecture, and Sections 3, 4, and 5 detail the main benefits of this architecture: extensibility, security, and transparency. Section 6 gives an evaluation of the implementation in terms of code size and performance. Finally we discuss related work, primarily regarding remote execution, and conclude.

2 Architecture

REX is designed to work with the Self-certifying File System (SFS), a secure, global network file system. SFS provides REX's user and server authentication facilities. REX also shares SFS's 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. Further, the use of local file descriptor passing allows REX to be broken into small functional units, minimizing the amount of privileged code.

The REX architecture offers extensibility through a communication abstraction that connects remote code (including arbitrary user programs) through the familiar interface of file descriptors. These pieces of code are called *modules*. REX groups file descriptors into *channels*, and channels into *sessions*. A *session* corresponds to all cryptographically protected communication over a single TCP connection between a REX client and a particular server. For each pair of communicating modules, there exists a *channel* within some session. Each channel can contain an arbitrary number of *file descriptor* pairs, over which modules may send data or more file descriptors.

Sections 2.1 and 2.2 provide background on user authentication and local and remote file descriptor passing. Sections 2.3 and 2.4 describe how REX establishes new sessions and how the channel abstraction is used to connect modules. Finally, this section concludes with a discussion of connection caching.

2.1 User Authentication in SFS

The key SFS subsystem that REX leverages is the user authentication infrastructure, which consists of two programs. The first is a per-user agent process, *sfsagent*,

which runs on the client machine. The agent stores a user's private key and signs authentication requests on his behalf. The second program is an authentication server, *sfsauthd*, which runs on the server machine. The authentication server verifies the signatures on authentication requests and then maps user public keys to local Unix accounts based on a database of registered SFS users.

2.2 File descriptor passing

File descriptors are numerical handles which name an opened file, socket, device, or other file-like resource. Most I/O in Unix is performed by reading from and writing to file descriptors. Unix also provides a facility for passing a file descriptor to another process through the *sendmsg* and *recvmsg* system calls on Unix-domain sockets [23].

REX uses local file descriptor passing between daemons, particularly on the server. This mechanism makes it easy to split functionality at a connection endpoint between a privileged and unprivileged process, typically by handing the connection from the privileged to the unprivileged process after some initialization phase. The use of local file descriptor passing as it relates to security is discussed further in Section 4.

REX also introduces the emulation of file descriptor passing between machines. This mechanism allows many extensions such as port forwarding and pseudo-terminal allocation to be implemented outside of the core system, thereby increasing extensibility. The use of file descriptor passing over the network is described in more detail in Section 3.

2.3 Sessions

Figures 1 and 2 show how REX establishes a session between a client machine (left) and a server (right). Boxes with a gray background are SFS programs that REX uses, while boxes with a white background are part of REX. Boxes with a filled upper-right corner are programs that run with superuser privileges. (The SFS agent is half-gray, half-white because even though it was part of the original SFS architecture, we extended it to support REX as described below.)

Setting up a REX session has two stages. In Stage I, the client establishes a secure, authenticated connection to the server. We call this initial connection the "master" REX session. In Stage II, the client creates new REX sessions, based on the master session, to run programs on the server.

2.3.1 Stage I

The user invokes the *rex* client[1] in order to start a new REX session. First, *rex* contacts the user's agent and asks it to establish a session to the desired server (Figure 1, 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]

Mazières et al. [20] describe several mechanisms through which the client can obtain the server's key. By default, REX, like SSH, maintains a cache of server public keys that it has already seen. REX, however, avoids possible man-in-the-middle attacks when contacting a server for the first time by using the Secure Remote Password (SRP) protocol [35].

Next, the *sfsagent* authenticates its user to *rexd* (Step 3). The agent signs an authentication request, which it passes to the server through the secure connection. *Rexd* passes the authentication request to the authentication server, *sfsauthd*, which verifies the signature and identifies the user (maps the user's public key 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). *Rexd* uses file descriptor passing to hand the secure connection to *proxy*, which processes remote execution requests from the user (Step 5). The *sfsagent* maintains a connection to *proxy* in order to keep this master REX 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. The master REX session is the basis for subsequent sessions between this user and server.

2.3.2 Stage II

To run a program on the server, the *rex* client notifies the user's *sfsagent* that it wants to create a new session to the given server (Figure 2, Step 1). *Sfsagent* looks up the corresponding master REX session and hands *rex* session keys for a new session to the same *proxy*. *Rex* then connects to *rexd* (Step 2). *Rexd* checks that *rex* indeed possesses appropriate keys, and if so hands the connection off to *proxy* through file descriptor passing (Step 3). Finally, *rex* asks *proxy* to spawn a program, say /bin/ls, with a certain number of file descriptors (Step 4). Rex then mediates the exchange of data between these file

---

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

[2]Since the SFS file server, authentication server, and *rexd* all listen on the same TCP port, connection setup by default also goes through an *sfssd* "meta-server." *Sfssd* demultiplexes incoming connections and hands them off to the appropriate daemon using file descriptor passing.
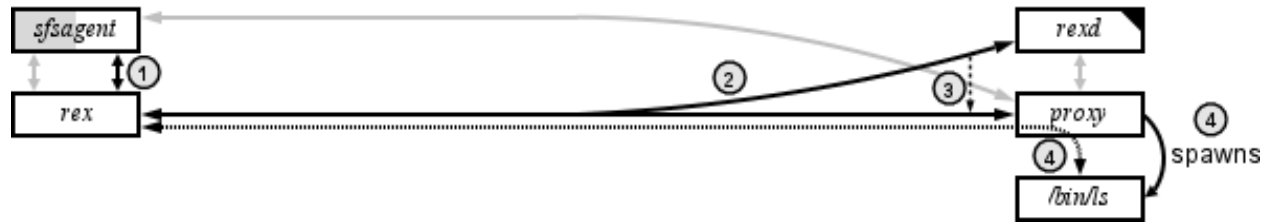
Figure 1: Setting up a REX session (Stage I)

Figure 2: Setting up a REX session (Stage II). Gray lines represent connections that were established during Stage I.

descriptors and a component on the client side with a *channel*.

2.4 Channels

The REX channel abstraction allows a pair of modules on different machines to communicate as if they were running on the same machine, connected by one or more Unix-domain sockets. When the client module writes data to a file descriptor, *rex* encapsulates that data as an RPC and sends it to *proxy*, which in turn unpacks the data and writes it to the appropriate file descriptor. The server module can then read the data on its corresponding file descriptor. *Proxy* similarly relays any data it reads back to *rex*.

The client creates channels through an RPC that specifies the name of the server module to run, a set of command line arguments and environment variables to set, and the number file descriptors the spawned module should inherit. (If fewer than three file descriptors 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.

Channels are the mechanism through which REX emulates file descriptor passing over the network. When a module passes a file descriptor to *rex*, *rex* notifies *proxy* through an RPC. *Proxy* then creates a new Unix-domain socket pair, passes one end to the local module, and allo-

cates a new file descriptor number within the channel for the other end. Conversely, when a module passes a file descriptor to *proxy*, *proxy* allocates a new file descriptor number for it within the appropriate channel and notifies *rex*, which similarly passes one end of a new socket pair to the local module. As Section 3 demonstrates in detail, this emulated file descriptor passing is the foundation of REX's extensibility.

2.5 Connection caching

The REX protocol naturally lends itself to *connection caching* [6, 10]. Because *rex* uses the *sfsagent* to establish a master session with *rexd/proxy* first, the *sfsagent* can remember (maintain) that connection and use it to set up subsequent REX sessions quickly. The initial REX connection to a remote machine is set up using public-key cryptography. Once this connection is established, REX uses symmetric cryptography to secure communication over the untrusted network. Subsequent REX connections to the same machine can 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 *sfs-*

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

Figure 3: *Sfsagent* and *rexd* use the *MasterSessionKey*s and sequence number (*i*) to compute new *SessionKey*s.

agent. A utility *sfskey* lets the user list and manage open connections.

Once the master session has been established, the *rex* client can create subsequent secure connections (sessions) to the same server using the following protocol. First, *rex* contacts the *sfsagent* and requests a new session. The agent computes the values shown in Figure 3 based on the *MasterSessionKey*s (one for each direction) that were established using public-key cryptography during the initial connection. The *SessionKey*s are the symmetric keys that the *rex* client uses to encrypt its connection to *proxy*. They are computed as the HMAC-SHA-1 [7, 17] of a sequence number *i* keyed by the *MasterSessionKey*s. The agent generates a unique sequence number for each REX connection to prevent an adversary from replaying old REX sessions. The *SessionID* is a SHA-1 [7] 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 insecure connection to *rexd* 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 cached connection based on the *MasterSessionID*. Then, *rexd* computes the *SessionKey*s and the *SessionID* for the new REX session (as in Figure 3) based on the sequence number that it just received and the *MasterSessionKey*s that it knows from the initial connection by the *sfsagent*. *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. Finally, *rex* and *proxy* both begin securing (encryption and message authentication code) the connection.

After *rex* and *proxy* establish a secure REX session, the *rex* client can create a new REX channel as described above. *Proxy* (and possibly also *rex*) will spawn the appropriate modules which can now communicate securely over the network. Subsequent connections proceed in the same way, allowing REX to rapidly execute processes on the server.

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. REX offers these features and also provides users with an interface to add new ones. REX's extensibility stems primarily from a single abstraction: 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.

3.1 TTY Support

REX provides pseudo-terminal support to interactive login sessions using the channel abstraction and file descriptor passing as follows. The *rex* client tells *proxy* to launch a module called *ttyd*, which takes as an argument the name of the actual program that the user wants to run. Typically, for remote login, the argument to *ttyd* is the user's shell.

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 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 own-

---

[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.

ership 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 (e.g., the user's shell). It spawns the process with the slave side of the TTY as its standard file descriptors and controlling terminal. Then, *ttyd* sends the file descriptor of the TTY's master side back to the *rex* client via the REX channel. On the client machine, *rex* copies data back and forth between this copy of the TTY's master file descriptor and the local terminal (e.g., the *xterm* in which *rex* was started).

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*, and sends the new window size over the channel using file descriptor 0. Upon receiving the window resize message, *ttyd* updates the server side pseudo-terminal through an *ioctl*.

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*, which 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.

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 channels described above, the *rex* client itself does not act as the client module. This generic mechanism allows 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 over the channel. The *moduled* module on the server is a wrapper program that reads a file descriptor from its standard input and spawns *connect* with the 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

---

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

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.

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, intramachine IPC makes use of a 120-line setgid program, *suidconnect*. *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.

3.5   File system integration

One of the main motivations for building REX was to provide a remote execution tool that was integrated tightly with the SFS file system. When a user logs into a remote machine, he should see the same file systems as on the local machine. REX achieves this behavior by forwarding the *sfsagent*, which maintains a per-user view of the /sfs directory. Additionally, because the agent handles all of the configurable aspects of a user's environment—server key management, user authentication, revocation—the remote login session acts the same as the local one. 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.

4   Security

The REX architecture provides two 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.

4.1   Minimizing exploitable code

In recent years, remote exploits have become a major concern for software developers. Buffer overruns and other bugs have led to serious system security compromises. 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* as that user. *Rexd* uses local file descriptor passing to pass the client connection to *proxy*.

REX also tries to avoid 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. These 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.

4.2   Managing trust policies

One particularly difficult issue with remote login is the problem of accurately reflecting users' trust in the various machines they log into. For example, a user may use local machine *A* to log into remote machine *B*, and then login from that session on *B* back to *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 simply type their passwords. This is both less convenient and less secure, as an 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 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 be-

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

ing 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. Moreover, 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) 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.

5 Transparency

Due to the limited size of the IPv4 address space, machines often do not have static, globally routable network addresses. When an organization has more computers than IP addresses, it must typically resort to Network Address Translation, or NAT. With NAT, machines have private [25] (not globally routable) IP addresses on the local network, and a gateway re-writes the source address of any outgoing packets to be globally routable. The gateway then inverts this translation on any incoming packets, so it can deliver them to the right port on the appropriate local machine.

While NAT gateways let clients with private IP addresses connect normally to external machines, they have no analogous way of transparently supporting incoming connections to local servers. The reason is that most servers listen on well-known TCP or UDP ports. If the number of servers exceeds the number of globally routable IP addresses available, multiple server machines must share the same IP address, requiring some form of application-specific demultiplexing.

A related problem is that of dropped TCP connections. Sometimes the only globally-routable IP address available to a machine (or network of machines) is temporarily assigned and periodically changes. Also, laptops usually need to change IP addresses when transported between buildings. If one end of a TCP connection changes its IP address, the entire connection must be aborted.

NAT is another source of aborted TCP connections. Because NAT gateways must keep state for every active TCP connection, they can prematurely terminate a TCP connection when rebooted or when purging state entries for other reasons. Some NAT implementations (notably some cheap home routers optimized for web browsing) aggressively terminate TCP connections after only a few minutes of idle time.

Dropped TCP connections are particularly annoying with traditional remote login tools, as they cause the user's entire session to be aborted. Sessions may abort at inopportune times, when users are in the middle of editing files. Moreover all state associated with a dropped session is typically lost, including GUI windows forwarded from the remote machine.

Several design features allow REX to operate transparently through NATs and without fixed IP addresses. First, the SFS connection protocol allows servers to share IP addresses and even TCP ports, so that clients can connect transparently to arbitrarily many servers behind a NAT gateway with a single globally-routable IP address. Second, REX supports transparent resumption of aborted TCP connections [28, 39], so that a session need not be restarted after a change of IP address or NAT state flush.

5.1 Address and port sharing

The SFS framework, into which REX fits, provides two solutions for configuring servers behind NATs. The first approach, which we call address sharing, is to assign each internal SFS server a unique TCP port number. Most NAT gateways can be configured to have static mappings of external port numbers to private addresses and port numbers. For instance, TCP port 600 on the external IP address might always be translated to TCP port 4 of internal IP address *A*, while external port 601 is always mapped to port 4 on internal address *B*.

Though SFS servers by default listen on TCP port 4, a different port number can be specified with DNS SRV [12] records. Each SRV record maps an SFS server name and service to a server hostname (i.e., the name of the globally-routable IP address), a port number, and some priority information (so that multiple SRV records can be used for load balancing). In this way, the NAT administrator can configure an external TCP port for each internal SFS machine, and publish port numbers through DNS. External clients will then transparently connect to the appropriate port of the external address.

A second approach, which we call port sharing, requires only a single external TCP port number for all internal servers. All SFS protocols, including REX, begin with a CONNECT RPC in which the client specifies the desired self-certifying server name and service type (e.g., REX, file system, or authentication server). SFS's "meta-

```
***** 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

[OK]

Figure 4: A GUI confirmation program

server" program, *sfssd*, can proxy TCP traffic to different internal IP addresses based on the contents of the initial CONNECT RPC. Port sharing with *sfssd* is similar to using the HTTP `Host` header with an HTTP proxy.

One advantage of port sharing is that *sfssd* can be configured to proxy certain services for a given internal server but not others (e.g., exporting an SFS file server but disallowing remote logins to it through REX). A security-conscious gateway administrator therefore has better control over what services are being made externally available. The disadvantage of port sharing is that its user-level TCP proxying consumes more CPU time and adds more latency than typical in-kernel NAT implementations.

A final issue with NATs is that, for efficiency reasons, machines on the internal network should connect to each other without going through the NAT gateway. The best way to achieve this goal is to run a split DNS server, which for the same hostname serves internal addresses to internal clients and external addresses to external clients. BIND and several other popular DNS servers support such functionality, but a number of users on the SFS mailing list have complained of the complexity of configuring DNS servers. Therefore, if split DNS is not available, DNS records can be set to point to the external IP address and internal machines can use a file `/etc/sfs/sfs_hosts` to override DNS with internal addresses. This file's syntax is a superset of traditional `/etc/hosts`, extended to allow port numbers to be specified.

5.2 Session resumption

When a TCP connection aborts, REX provides the ability to resume the session over a new TCP connection. In order not to increase the amount of trusted or remotely exploitable code, this functionality is implemented entirely in *proxy*, with no changes required to *rexd*. To resume an aborted TCP connection, the client first attaches to *proxy* through *rexd*, using a new sequence number. It then issues a RESUME RPC, supplying the sequence number of the old session. This RPC causes the *proxy* to delete the state of the current session and replace it with that of the old session.

REX uses a bi-directional RPC protocol. Any input to *rex* prompts it to send an RPC to *proxy*, and similarly any program output to *proxy* results in an RPC to *rex*. For a resumable connection, *rex* and *proxy* each keep a replay cache of recently transmitted RPCs replies. Resumption then just consists of replaying all unanswered RPCs. In order to determine when something can be evicted from the replay cache, the RPC code conservatively determines when the other side has received a reply based on two factors: the size of the kernel's TCP send buffer and replies to RPCs in the other direction.

One issue introduced by session resumption is the potential to leave stale proxies around if *rex* processes are terminated. REX employs several techniques to reduce the incidence of stale proxies. First, each *rex* client maintains a connection to the user's agent. If a resumable *rex* process dies (for instance because the user terminates it with the Unix "`kill -9`" command), the agent detects

this fact by an end-of-file, and informs the remote proxy that the particular session can be garbage-collected.

Second, each agent has a unique identifier, based on the user's login name and the name of the machine it is running on. The agent's identity is supplied as a command line option to *proxy* (which, in particular, makes it visible through the Unix `ps` command). Whenever *proxy* is launched with a particular agent identity, it informs any previous *proxy* running with the same identity though a named Unix-domain socket in `/tmp`, and the previous proxy then considers all sessions non-resumable. In the event that the agent ungracefully exits (for example, the client crashes and reboots), this mechanism causes the old proxy to exit the next time the user logs into the same server.

Session resumption works transparently even when the server changes IP address, so long as the server publishes its current address through DNS (e.g., using some sort of dynamic DNS scheme like `dyndns.org`). However, there are some subtleties to making this work properly because of the fact that DNS can also be used for load balancing—for instance, a hostname like `dialup.mit.edu` might actually point to a pool of login servers. In such cases, when a client changes IP address, it must resume its REX session to the same dialup server. To achieve this, REX revalidates all DNS information when reconnecting, and chooses the same DNS records as for the initial connection if still available. More precisely, when the original connection used an SRV record, if the particular hostname and port chosen the first time are still available, reconnection uses them again. For a given hostname, if the particular IP address initially used is still available, reconnection again re-uses it.

We note that the level of indirection provided by SRV records allows the location of an entire network of servers behind a NAT gateway to be updated with the change of a single DNS A (address) record. For example, Figure 5 shows an example of SRV records for four SFS servers in the static DNS domain `mydomain.org`, located behind a NAT gateway called `mynat.dyndns.org`. If the gateway's external address changes, only the DNS record of `mynat.dyndns.org` needs to be updated—the `mydomain.org` domain can remain unchanged.

6 Evaluation

First, this section quantifies REX's extensible architecture in terms of code size. Second, we compare the performance of REX with the OpenSSH [21] implementation of SSH protocol version 2 [37]. The measurements demonstrate that the extensibility gained from file descriptor passing comes at little or no cost.

6.1 Code size

REX has a simple and extensible design. Its wire protocol specification is only 230 lines of Sun XDR code [29]. REX has two component programs that run with enhanced privileges. *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 [19]). *Ptyd* allocates pseudo-terminals to users that 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 constitute less than 900 lines of code.

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.

6.2 Performance

We measured the performance of REX and OpenSSH 3.8p1 [21] 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 uses the Rabin-Williams cryptosystem [33] 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 [14] cipher for confidentiality. For integrity, SFS uses a SHA-1-based message authentication code while SSH uses HMAC-SHA-1 [7, 17]. Our SSH server had the privilege separation feature [24] enabled.

6.2.1 Remote login

We compare the performance of establishing a remote login using REX and SSH. We expect both SSH and REX to perform similarly, except that REX should have

```
;   SERVICE/NAME                       PRIO/WGHT  PORT SERVER
_sfs._tcp.server-a.mydomain.org.  SRV   0   1    600  mynat.dyndns.org.
_sfs._tcp.server-b.mydomain.org.  SRV   0   1    601  mynat.dyndns.org.
_sfs._tcp.dialup.mydomain.org.    SRV   0   1    602  mynat.dyndns.org.
_sfs._tcp.dialup.mydomain.org.    SRV   0   1    603  mynat.dyndns.org.
```

Figure 5: An example of DNS SRV for four SFS servers on different TCP ports of `mynat.dyndns.org`. Such configurations are useful when `mynat.dyndns.org` is a NAT gateway, forwarding different TCP ports to different internal server machines. The *priority* and *weight* columns affect load balancing across duplicate records. The values are meaningless for `server-a` and `server-b`, and for `dialup` result in uniform distribution of connections across TCP ports 602 and 603 of `mynat.dyndns.org`.

a lower latency for subsequent logins because of connection caching.

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

Table 1: Latency of SSH and REX logins

Table 1 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 slightly 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 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 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. Were SSH to implement connection caching, we would expect performance similar to REX's on subsequent logins.

6.2.2 Port forwarding throughput

Both SSH and REX can forward ports and X11 connections. To demonstrate that REX performs just as well as SSH, we measure the throughput of a forwarded TCP port with NetPipe [27]. NetPipe streams data using a variety of block sizes to find peak throughput.

Protocol	Throughput	Latency
TCP	87.1 Mbit/sec	118 μsec
SSH	86.2 Mbit/sec	294 μsec
REX	86.0 Mbit/sec	394 μsec

Table 2: Throughput and latency of TCP port forwarding

We first measure the throughput of an ordinary, insecure TCP connection. Table 2 shows that the maximum TCP throughput is 87.1 Mbit/sec. The round-trip latency represents the time to send one byte of data from the client to the server, and receive acknowledgment. Next, we measure the throughput of a forwarded port over established SSH and REX connections. Table 2 shows 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.

7   Related Work

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.

7.1 SSH

SSH [38] is the de-facto standard for secure remote execution and login. SSH is decentralized: one needs only local superuser privileges to run the SSH server daemon, and one does not need to obtain server certificates or otherwise register with any sort of realm administrator in order to connect to the SSH server. SSH also offers several modes of user authentication. For example, it has optional support for Kerberos [30], allowing password-free login plus ticket and AFS [13] token forwarding.

SSH was the main inspiration for REX, as we needed an SSH-like tool that could work with SFS. Though we could have extended SSH for the task, we decided to build REX from scratch for several reasons. First, we believed a design based on file descriptor passing would simplify implementation, improve security, and increase extensibility. Leveraging SFS's RPC compiler and library further reduced the amount of new code needed. We also wished to take advantage of SFS's infrastructure for user and server authentication, particularly its use of SRP to sidestep potential man-in-the-middle attacks. Finally, as commonly configured, SSH servers read files in users' home directories before authenticating them, which is inconvenient when the home directories themselves reside on SFS.

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 support for NATs 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.

We believe many of the ideas in REX are applicable to SSH and other remote login tools, and hope that SSH and REX can increasingly adopt each other's features. For example, as part of the privilege separation code in OpenSSH [21], the OpenSSH server internally handles pseudo-terminals with file descriptor passing. 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.

7.2 Kerberos

Kerberos [30] is a centralized authentication system which includes remote login and execution utilities. It provides a unified way of naming, authenticating, and authorizing principals. Kerberos organizes users and machines into realms. Joining an existing realm (i.e., setting up a server) requires permission from and coordination with that realm's trusted administrator. In part because Kerberos is based on shared-secret cryptography, creating a new realm is not a simple task and requires administrative permission to interoperate with existing realms.

Kerberized remote login is based on this 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. The AFS [13] file system uses Kerberos for authentication, and Kerberized remote login can authenticate users to the file system before logging them in. REX provides similar support for the SFS file system.

7.3 Globus

The Globus [8] Project provides a Grid metacomputing infrastructure that supports remote execution and job submission through a resource allocation manager called GRAM [5] and access to global storage resources through GASS [1]. 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 [11]. 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.

The Grid Security Infrastructure (GSI) [3, 9] provides security and authentication to Grid-based services. GSI is based on X509 [36] public-key certificates and the SSL/TLS protocols [6]. Recent extensions to GSI add support for proxy certificates [32], 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, similar to REX's selective signing mechanism.

7.4 Secure rlogin

Before SSH, researchers explored other options for secure remote login [16, 31]. Kim et al. [16] 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, similar to IPSec [15].

7.5 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 SSH agent, for example, is capable of 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. By contrast, 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.

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 indepently 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.

Recently, the security architecture for the Plan 9 system has been redesigned [4]. The new Plan 9 architecture has an agent, *factotum*, which is similar to the SSH and SFS agents but is implemented as a file server.

The Taos operating system [18, 34] and the Echo file system [2] also have notions of an authentication agent. Unlike SFS, they both implement the agent as an operating-system component rather than as a user-controlled program.

7.6 File descriptor passing

An alternative to file descriptor passing is file namespace passing, as is done in Plan 9 [22]. 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.

8 Conclusions

REX provides secure remote login and execution in the tradition of SSH. REX offers a new architecture with three main goals—extensibility, security, and transparent connection persistence in the absence of global routing. REX's extensibility, based on emulated file descriptor passing between machines, allows users to add new functions to REX without changing the protocol. REX's security benefits are a limited amount of exploitable code and a convenient mechanism for building trust policies. Finally, REX provides transparent operation in today's complex network configurations, which include NAT and dynamic IPs.

The current REX implementation demonstrates that the REX architecture is viable. We hope that the new ideas upon which REX is built will find wider applicability in other systems. REX is available as part of the SFS distribution (`http://www.fs.net/`).

9 Acknowledgments

References

[1] 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.

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

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

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

[5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. 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, 1998.

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

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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, 1999.

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

[22] 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, Apr 1993.

[23] Dave Presotto and Dennis Ritchie. Interprocess communication in the eighth edition UNIX system. In *Proceedings of the 1985 Summer USENIX Conference*, Portland, OR, 1985.

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

[25] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets. RFC 1918, Network Working Group, February 1996.

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

[27] 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.

[28] Alex C. Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, December 2002.

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

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

[31] 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.

[32] V. Welch, I. Foster, C. Kesselman, O. Mulmo, S. Tuecke L. Pearlman, J. Gawor, S. Meder, and F. Siebenlist. X.509 proxy certificates for dynamic delegation. In *Proceedings of the 3rd Annual PKI R&D Workshop*, April 2004.

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

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

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

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

[37] 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.

[38] 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.

[39] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, GA, September 2002.