REX: Secure, modular remote execution through file descriptor passing

Michael Kaminsky, Eric Peterson, Kevin Fu, David Mazières, M. Frans Kaashoek
MIT Laboratory for Computer Science
NYU Department of Computer Science
{kaminsky, ericp, fubob}@lcs.mit.edu,
dm@cs.nyu.edu, kaashoek@lcs.mit.edu

Abstract

The ubiquitous SSH package has demonstrated the importance of secure remote login and execution. This paper presents a new system, REX, designed to provide remote login and execution in the context of the SFS secure distributed file system. REX departs from traditional remote login design and is built around two main mechanisms—file descriptor passing and a user agent process.

File descriptor passing allows REX to be split into several smaller pieces; privileged code can run as its own process to provide enhanced security guarantees. REX also emulates secure file descriptor passing over network connections, allowing users to build extensions to REX outside of the core REX software.

REX uses and extends SFS's agent mechanism to provide a transparent distributed computing environment to users. The agent stores private keys, server nicknames, and other per-user configuration state; REX makes the SFS agent available to programs that it executes on remote machines.

We have an implementation of REX and demonstrate that its flexibility does not come at the cost of performance. Initial REX connections are comparable to those of SSH in speed, while subsequent connections are much faster because REX exploits the SFS agent to cache connection state to avoid costly public-key operations.

1 Introduction

Remote login and execution are network utilities 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. Users expect such features as TCP port and X Window System forwarding, facilities for copying files back and forth, cryptographic user authentication, transfer of user credentials across machines, integration with network file systems, server public key management, pseudo-terminals and more. Many of these features have special requirements, resulting in complicated, extensible remote login protocols to which people add new message types for new functionality.

This paper argues that the wide variety of features people need from remote login can actually be implemented through two simple abstractions—*file descriptor passing* and an extensible, general-purpose *user agent*. We have built a new remote login system called REX, part of the SFS [16] computing environment, based upon these two ideas.

First, REX harnesses and extends Unix's ability to pass file descriptors between processes over Unix-domain sockets. File descriptor passing facilitates privilege separation by allowing, for example, decomposition of the REX server into two components: a small trusted program, *rexd* (the core remote login server), and a slightly larger program, *proxy* (a per-user daemon process), which only authenticated users can spawn and communicate with. The latest versions of OpenSSH [17] have begun to move in this direction and have introduced local file descriptor passing to aid in pseudo-terminal support.

REX also emulates secure file descriptor passing between processes running on different machines. Intermachine file descriptor passing allows functions that are typically built into other remote execution tools, such as port forwarding, to be cleanly implemented outside of REX by a pair of entirely separate programs. One module, running on the server, listens for connections to a particular port (or even a Unix domain socket) and "passes" the accepted file descriptor through REX to another module, running on the client, which connects it to a local port (or Unix domain socket).

The second mechanism that REX uses to achieve its goals is an extensible, general-purpose user agent. REX uses the SFS agent, which, like the SSH agent, is a process that runs on behalf of the user. The SFS agent, however,

This research was supported by the DARPA Composable High Assurance Trusted Systems program (BAA #01-24), under contract #N66001-01-1-8927. Michael Kaminsky was supported by a National Science Foundation Graduate Research Fellowship.

holds all of a user's personal state (i.e., his "computing environment") and not just his private keys. The SFS agent can also store public keys of servers, nicknames of commonly used servers, and various trust policies. The agent maintains all of a user's state behind a simple Remote Procedure Call (RPC) interface, and thus can be "forwarded" to other hosts, providing users an identical environment wherever they log in. REX also allows selective signing for situations in which the user does not trust all of the remote machines he logs into with access to his agent.

We have implemented the REX remote execution facility, and the source code is freely available. REX only adds 800 lines of trusted code (not counting general-purpose crypto and RPC libraries) and provides an architecture for implementing extensions as separate programs. REX currently offers modules that handle pseudo-terminal support, TCP port forwarding, X11 forwarding with cookie authentication, and Unix domain socket forwarding. REX interoperates with SFS and the SFS agent to provide a secure, global computing environment. The remote execution tool is fast because REX caches connection state to avoid repeatedly executing the public-key cryptographic operations involved in authentication. REX has been bundled with the SFS distribution since release 0.6. The REX server is enabled by default in the current release of SFS.

This paper describes REX. Section 2 provides a brief introduction to SFS. Section 3 details REX's design and implementation, and Section 4 gives an evaluation of the implementation in terms of code size and performance. Finally, we conclude the paper with some related work, primarily regarding remote execution (Section 5).

2 Background

The goal of the SFS project is to secure the network communications comprising people's day to day computing. SFS consists of a network file system, the REX remote execution facility, a per-user *agent* process, and an authentication daemon. These components are illustrated in Figure 1.

Building any secure distributed system raises a number of questions: How do clients and servers communicate securely? How does a client know it is talking to the intended server and not an impostor? How are users authenticated to servers? What happens when cryptographic keys are compromised? In fact, no single set of answers is best for every situation. Thus, SFS attempts to tease the questions apart in such a way that people can select and recombine the solutions most suitable to their needs.

At the lowest level, SFS provides secure client-server communication through the abstraction of *self-certifying hostnames*. A self-certifying hostname contains the hash of a server's public key (like a PGP [33] fingerprint). Given such a hash, an SFS client uses well-known cryptographic techniques to establish a secure channel to the server, encrypting and authenticating all messages to protect them from network eavesdropping and tampering attacks. (See Mazières et al. [16] for details.)

Unfortunately, self-certifying hostnames are not human-readable; most users will never want to see or type them. Moreover, someone who accesses the wrong name may be tricked into accessing a malicious server with an attacker's public key. Thus, the problem of ensuring users reach the right server boils down to making sure they access the right self-certifying hostname.

SFS provides several techniques for users to obtain self-certifying hostnames. The simplest is through a password. If, for example, a user types *rex myserver* and REX does not know *myserver*'s public key, REX will prompt the user for a password, use this password to authenticate the server to the client, and then securely download the server's self-certifying hostname. SFS employs SRP [30] for such password authentication.

One of the roles of the SFS agent is to manage these self-certifying hostnames and minimize the number of times users have to type passwords. Each user typically runs a single agent on his local machine, responsible for all local and remote REX invocations and file system accesses. In the above example, after obtaining a self-certifying hostname for *myserver*, REX informs the agent of the mapping from the human-readable name *myserver* to the corresponding self-certifying hostname. Subsequent invocations of REX can simply retrieve this mapping from the agent, so the user does not need to enter a password.

In addition to static mappings, the SFS agent allows arbitrary external *certification programs* to map human-readable names to self-certifying hostnames automatically. Whenever a user executes *rex* or makes a file system access to an unknown human-readable server name, the user's agent proceeds to execute its certification programs to determine the appropriate self-certifying name. The SFS agent contains a number of features for advanced users, including perl-style regular expression filters on programs. In practice, however, simple shell scripts make useful certification programs. For example, a 12-line shell script can implement the `known_hosts` mechanism built into SSH.

Another important function of SFS agents is to perform user authentication, convincing servers of the local user's identity. In this respect, SFS agents are similar to SSH [32] agents. The default user authentication mechanism uses public-key cryptography. The agent stores a copy of the user's private key, which it uses to sign authentication requests; the authentication server verifies the signature with a stored copy of the user's public key. The agent and authentication server are extensible and can support other authentication mechanisms such as shared key cryptography. User authentication is opaque to the file system and REX; changing the authentication mechanism only requires modification of the agent and authentication server.

One particularly important authentication mechanism is password authentication with SRP. As mentioned above, SRP authenticates servers to users, but it also simultane-

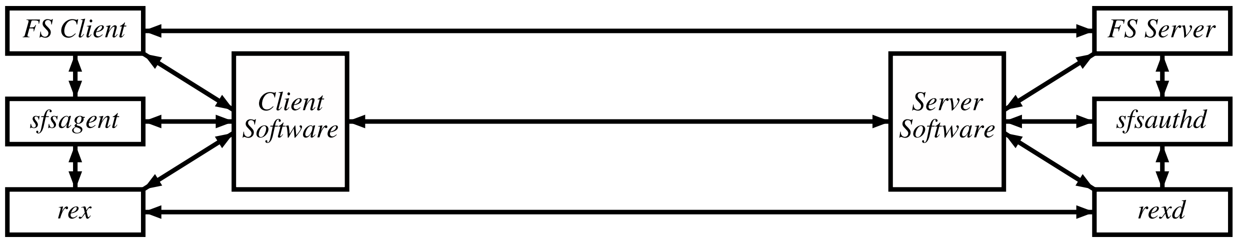

Figure 1: The major components of the SFS system.

ously authenticates users to servers. Thus, in the typical case that a user types a password, the password is used for mutual authentication and the user can immediately log into the server. Often the password additionally lets the user automatically decrypt his own private key, so that subsequent user authentication to any server in the same administrative realm can proceed without a password.

Finally, SFS agents handle *server revocation*, for cases when a server's private key has been compromised. Two mechanisms exist: the agent can maintain a list of revocation certificates which are self-signed by the compromised server key and the agent can maintain a list of fine-grain rules describing servers (self-certifying hostnames) that he does not want to access. A more detailed description of revocation is available in previous papers on SFS [14, 16].

3 Design and Implementation

The SFS and REX components communicate using Sun RPC [24], and the SFS libraries provide a secure transport for RPCs that travel over the network. Combined with file descriptor passing, RPC allows REX to be split into several small programs that can communicate in a clearly-defined way.

An RPC compiler generates the stubs that the programs use from a high-level specification. One can study the concise RPC specification file to understand the protocol. Furthermore, the RPC compiler's automatic stub generation avoids exploitable programmer errors, such as buffer overruns, that are common in message parsing code. Mazières [15] describes the RPC library that REX uses in more detail.

3.1 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 file descriptor passing—sending a file descriptor to a different process—through the *sendmsg* and *recvmsg* system calls on Unix domain sockets [19, 26].

REX uses local file descriptor passing between daemons, particularly on the server. This mechanism provides a simple and convenient way for a privileged program to perform a task on behalf of an unprivileged one and return the result to it in the form of a file descriptor. For example, once a user has been authenticated, the privileged (i.e., "root") REX server daemon, *rexd*, hands off the connection to a separate daemon, *proxy*, running as the user. The privileged *ptyd* daemon allocates pseudo-terminals and passes them to *ttyd* which runs with the privileges of a normal user. These privileged programs are small and only perform a single task to allow easy auditing.

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. For example, once *ttyd* receives a pseudo-terminal from *ptyd*, it passes one end of it to the REX client over the network (see Section 3.4).

3.2 Sessions

Figure 2 shows how REX establishes a new connection between a client machine (left) and a server (right); this connection is known as a REX *session*. Boxes with a gray background are SFS components that REX uses. Boxes with a white background are part of REX, and boxes with a filled upper-right corner are components which run with superuser privileges.

The *rex* client[1] is responsible for establishing the initial connection to *rexd* (Step 1).[2] First, the two processes set up a secure connection between their respective hosts using public-key cryptography (the details of setting up a secure SFS RPC connection are given by Mazières et al. [16]). Then, the *rex* client authenticates its user to *rexd* using the *sfsagent*. The agent signs an authentication request which it passes to the server through the secure connection. *Rexd* passes the authentication request to *sfsauthd*, which veri-

[1]This paper will use REX (capital letters) to refer to the remote execution facility as a whole and *rex* (emphasized 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. It can also be configured to proxy connections for other hosts, based on the server name requested by the client. This feature is useful for allowing remote login to multiple machines behind a NAT gateway with a single IP address.

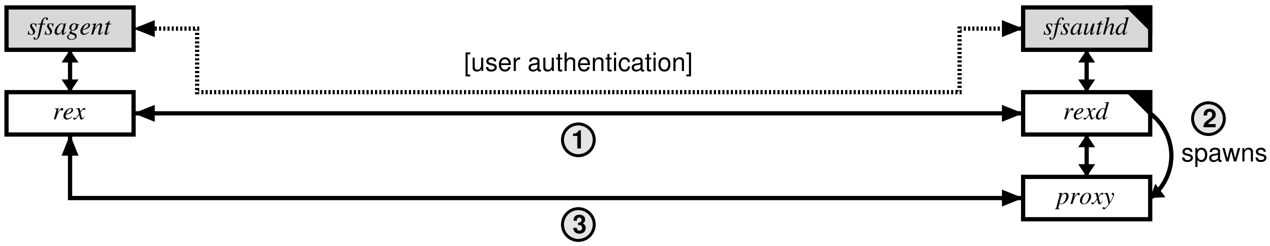

Figure 2: Setting up a REX session

fies the signature and identifies the user (maps the user's public key to a local account).

Once the initial connection has been established, *rexd*, which runs with the privileges of the superuser, spawns a new process called *proxy*, which runs with the privileges of the local user identified above (Step 2). *Rexd* uses file descriptor passing to hand the secure connection to *proxy*, which then assumes responsibility for the REX session (Step 3).

3.3 Channels

Within a REX session, the REX client can create one or more *channels*. Each channel provides a bidirectional data stream abstraction between two processes called *modules*. The REX channel allows these two modules to communicate as if they were running on the same machine with their standard file descriptors connected. If the client module sees data on a file descriptor, REX encapsulates that data as an RPC and sends it to *proxy*. *Proxy* unpacks the RPC and passes the data to the appropriate server module. The server module then sees the data on its corresponding file descriptor.

REX channels also support file descriptor passing for an arbitrary number of file descriptors. The server module, for example, can listen for network connections on a particular port. Upon accepting a new connection, the server module can pass that connection (i.e., the file descriptor which here is a socket) over the REX channel to the client module. The REX channel will proxy reads and writes to and from each end of this "remote socket pair."

The client creates channels with a special RPC which contains the name of the server module to run. *Proxy* receives the RPC and spawns the specified program. The client can then spawn a local module for the REX channel to connect up to the server module.

REX channels are implemented as C++ classes inside of the *rex* client. The rexchannel base class provides the basic channel functions, but the programmer can extend those functions to address more specific needs. A derived class represents the client side of a module pair. For convenience and efficiency, some of these derived REX channel classes handle the client end of the channel directly (inside of the *rex* client program itself); others spawn a separate program to serve as the client module.

One of the simplest channels in REX—derived from rexchannel—is called an execchannel. Figure 3 shows the execchannel setup. Here, *proxy* launches a program specified by the user (Step 4), and the REX channel connects that program's three standard file descriptors to the *rex* client's standard file descriptors (Step 5). In this case, the *rex* client itself (specifically, the execchannel C++ class) acts as the client module. The execchannel is analogous to SSH's ability to run a remote command without TTY support.

3.4 Using channels to support TTYs

REX provides pseudo-terminal support to interactive login sessions with ttychannel (see Figure 4). This channel uses file descriptor passing to offer TTY support to programs. 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 as the regular, unprivileged 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 to return both the master and slave sides of the TTY. *Ttyd* connects to *ptyd* with `suidconnect`—SFS's authenticated IPC mechanism (described further in Section 3.6). This mechanism lets *ptyd* securely track and record which users own which TTYs.[3] After receiving the TTY file descriptors, *ttyd* keeps its connection open to *ptyd*. Thus, when *ttyd* exits, *ptyd* detects the event with an end-of-file. *Ptyd* then revokes (with the *revoke* system call) any TTYs belonging to the terminated *ttyd*, cleans up device ownership and `utmp` entries, and recycles the TTYs.

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 connects the slave side of the TTY to that program's standard file descriptors. Then, *ttyd* sends

[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.

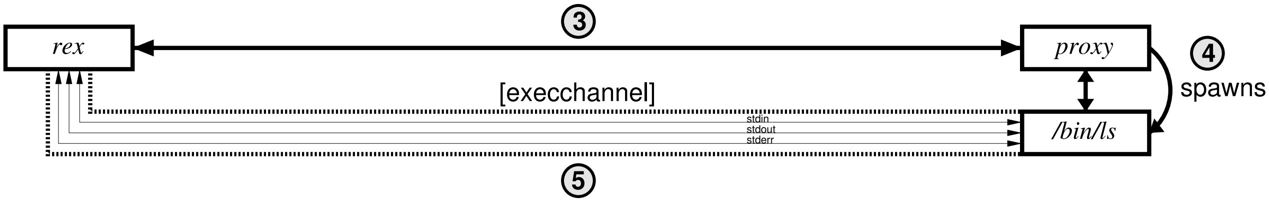

Figure 3: Setting up an execchannel

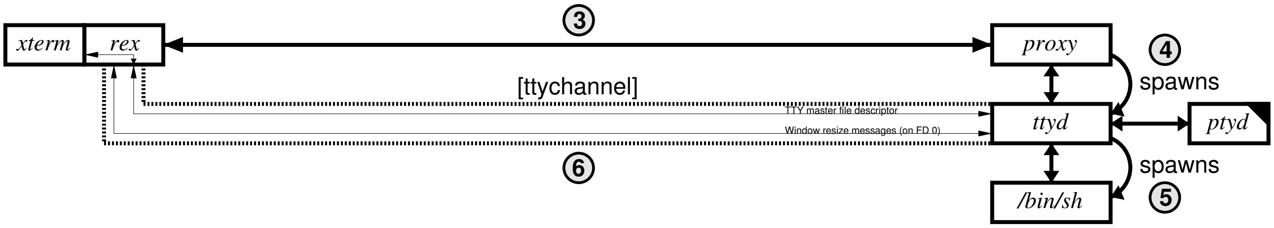

Figure 4: Setting up a ttychannel

the file descriptor of the TTY's master side back to the *rex* client via the ttychannel. On the client machine, *rex* connects the TTY's master file descriptor to the local terminal's file descriptor (e.g., the *xterm* that the *rex* client is running in).

The ttychannel, *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 ttychannel using file descriptor 0. Upon receiving the window resize message, *ttyd* updates the server side pseudo-terminal through an *ioctl*.

3.5 Using channels to support arbitrary programs

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

TCP port forwarding. Port forwarding essentially copies a port on one machine to another (see Figure 5). Connections to this copy are no different from direct connections to the original port. For example, a person might want to forward connections to port 8888 on a remote machine over a secure REX channel to his local Web server listening on port 80. REX provides these functions through binmodulechannel and three short, new programs: *listen*, *moduled* and *connect*. The *rex* client invocation is simply `rex -m 'moduled connect localhost:80' 'listen 8888' host`.

The *listen* module runs on the server and waits for connections to port 8888; upon receiving a connection, *listen* passes the accepted file descriptor over the binmodulechannel. The *moduled* module on the client 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 80 on the local machine and copies data between its standard input/output and the port. A client connecting to port 8888 on the server machine will effectively be connected to the Web server listening on port 80 of the client 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. A user does not need built-in support for such an extension but can add it using binmodulechannel. The *rex* client command syntax is similar to the port forwarding example: `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 file descriptor to the Unix domain socket named

[4] In practice, one would normally hide this socket in a protected directory.

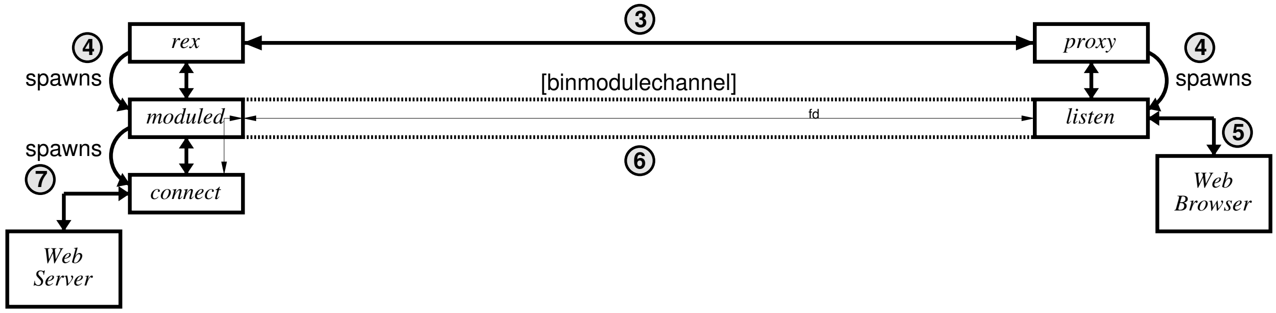

Figure 5: Setting up a binmodulechannel to forward a TCP port for Web proxying

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 sets `SSH_AUTH_SOCK` to be `/tmp/ssh-agent-sock`. When an SSH program makes a request of the SSH agent, the request is forwarded through the REX channel to the real agent running on the client machine. We have written a shell-script wrapper that hides these details of setting up SSH agent forwarding.

3.6 Using channels to forward the SFS agent

When first starting up, the *sfsagent* program connects to the local file system daemon to register itself using authenticated IPC. SFS's mechanism for authenticated, intra-machine 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. 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* through REX 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. Because REX uses the same agent as SFS, users see the same file systems during remote login. SSH differs from this architecture in that an SSH user's environment might depend on the contents of his `.ssh` directory on the remote machine.

3.7 Connection caching

REX provides fast remote execution through connection caching. 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 phase 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 computation 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*. A utility *sfskey* lets the user list and manage open connections.

REX eliminates the public-key operations on subsequent logins by caching the connection. The *sfsagent* and *rexd* keep track of the connection on the client and server machines, respectively. *Rex*, *sfsagent*, *rexd*, and *proxy* participate in the protocol described below to negotiate a new session key for each new REX connection to that server.

Figure 6 shows how REX sets up a new cached connection. Instead of connecting directly to *rexd*, the *rex* client asks the *sfsagent* to connect on its behalf. The agent uses public-key cryptography to establish a secure, authenticated connection to *rexd*. *Rexd* spawns *proxy* as before and passes off the connection to it. The *sfsagent* maintains a connection to *proxy* to prevent *proxy* from exiting—normally *proxy* will exit once all of the client connections have terminated. With connection caching, *proxy* remains running until the user explicitly terminates his REX session (using the *sfskey* command) or kills his *sfsagent*.

Once an initial connection has been established to a server, the *rex* client can make subsequent secure connections to that server as follows (see Figure 7). First, *rex* contacts the *sfsagent* and requests the *MasterSessionKey* and a new *SequenceNumber* for the connection (Step 1). The *MasterSessionKey* was established using public-key cryptography during the initial connection made by the *sfsagent*. The *SequenceNumber* is unique for each REX session (connection).

The *rex* client uses the *MasterSessionKey* and the *SequenceNumber* to compute the values listed in Figure 8. *SessionKey* is the symmetric key that the *rex* client uses to encrypt its connection to *proxy*; it is computed as a SHA-1 hash [4] of the *MasterSessionKey* and the *SequenceNum-*

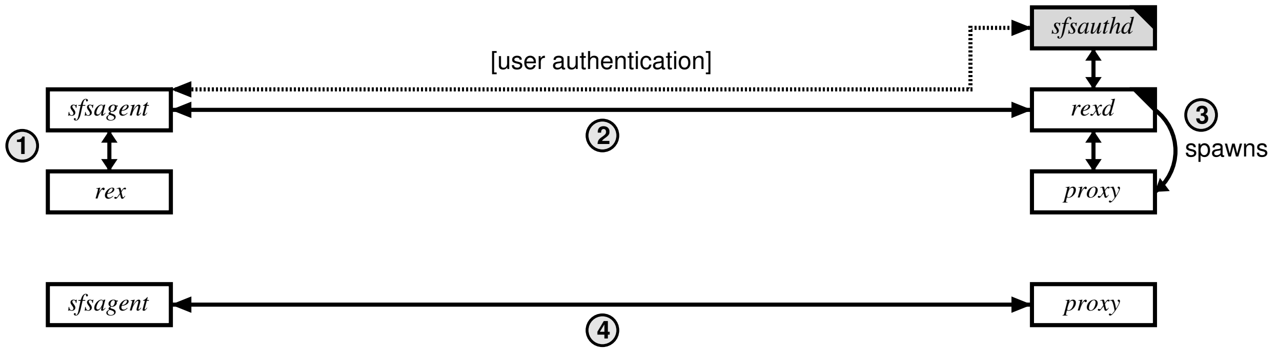

Figure 6: Setting up a cached REX session (initial connection)

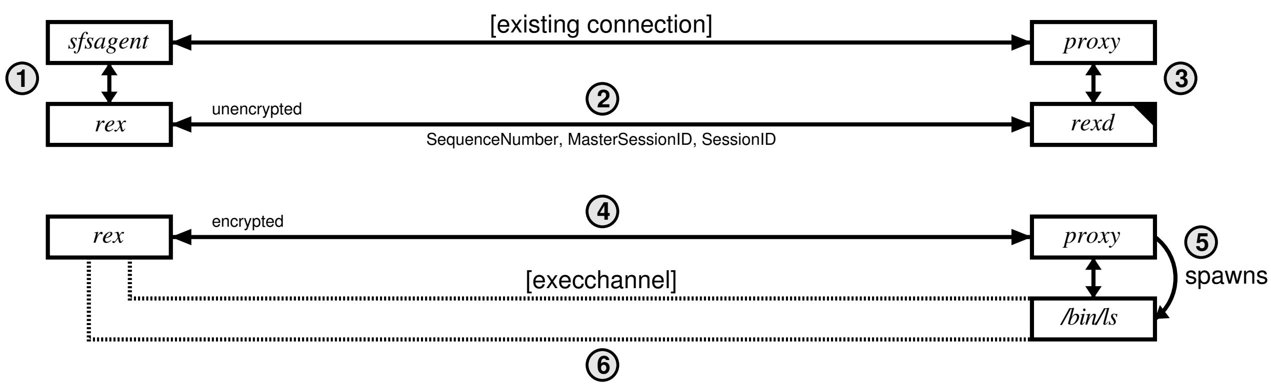

Figure 7: Setting up a cached REX session (subsequent connections)

$$
\begin{aligned}
SessionKey &= \mathrm{SHA}-1(MasterSessionKey, SequenceNumber) \\
SessionID &= \mathrm{SHA}-1(SessionKey) \\
MasterSessionID &= SessionID(SequenceNumber = 0)
\end{aligned}
$$

Figure 8: The *rex* client computes a new *SessionKey* from the *MasterSessionKey* and *SequenceNumber*.

ber. The *SequenceNumber* prevents an adversary from replaying old REX sessions. The hash also makes it infeasible to derive different *SessionKey* values from each other. The *SessionID* is a hash of the *SessionKey*, and the *MasterSessionID* is a hash of the *SessionID* where the *SequenceNumber* is 0.

Once the *rex* client computes these values, it makes an insecure connection to *rexd* (Step 2). It sends *rexd* the *SequenceNumber*, 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* and the *SessionID* for the new REX session based on the *SequenceNumber* that it just received and the *MasterSessionKey* 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* (Step 3). Finally, *rex* and *proxy* both begin securing (encryption and message authentication code) the connection (Step 4).

After *rex* and *proxy* establish a secure REX session, the *rex* client can create a new REX channel as described above (Steps 5 and 6). *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.

3.8 Agent forwarding and connection caching

REX caches connections in the *sfsagent* based solely on the identity of the remote machine. When REX forwards the *sfsagent*, these cached connections become available even in the remote login session. The user can issue the *rex* command from any machine that has access to his agent and benefit from the centrally located cache.

For example, if a user logs in from machine *A* (running the agent) to *B*, and then from *B* to *C*, REX forwards his agent to *B* and to *C*. This sequence of logins produces two cached connections: "to *B*" and "to *C*," both of which are stored in the *sfsagent* on *A*. Then, even though the user has never previously logged in from *A* to *C* directly, he can do so securely *without* using public-key cryptography because the his local agent on machine *A* and the *rexd* on machine *C* already share a *MasterSessionKey*. This technique provides a more efficient implementation, particularly for users that frequently move between a set of machines.

3.9 Selective signing

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 what if the user doesn't 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. During remote login, REX remembers the machines to which it has forwarded the agent. In the remote login session, when the user accesses an SFS file system, for example, his *sfsagent* will print out the name of the machine originating the authentication request and the machine to which the user is trying to authenticate. 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, signed authentication requests contain the name and public key of the server being accessed, as well as the particular service being accessed (e.g., REX or file system). Thus, 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 even allows users to plug-in arbitrary programs to ask their permission. Some users might want a GUI dialog box to pop up while others might simply prefer to respond to a text prompt. More advanced users can even register programs that automatically respond yes or no for certain groups of hosts.

4 Evaluation

First, this section quantifies REX's modular implementation in terms of code size. Second, we compare the performance of REX with the OpenSSH [17] implementation of SSH protocol version 2 [31]. The measurements demonstrate that the modularity gained from file descriptor passing comes at little or no computational cost.

4.1 Code size

REX has a simple and modular design. Its wire protocol specification is only 200 lines of Sun XDR code. REX has two component programs that run with enhanced privileges. *Rexd* receives incoming REX connections and adds only 400 lines of trusted code to a system (not counting the general-purpose RPC and crypto libraries from the SFS toolkit [15]). *Ptyd* allocates pseudo-terminals to users that have successfully authenticated and is also 400 lines of code.

Proxy, which runs with the privileges of the authenticated users is just over 700 lines of code and the *rex* client is 1,900 lines. Extensions to the *sfsagent* for connection caching constitute 600 lines of code.

Modules that extend REX's functions are also small. The *listen*, *moduled*, and *connect* modules which handle, among other things, TCP port forwarding and X11 forwarding are approximately 275, 50, and 400 lines of code, respectively. *Ttyd* is under 200 lines.

If REX were to gain a sizeable 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 pieces such as *proxy* or new external modules. Because of the modularity, well-defined interfaces, and the use of file descriptor passing, the trusted components will remain small and manageable.

4.2 Performance

We measured the performance of REX and OpenSSH 3.0.2p1 [17] on two 1.3 Ghz Athlon machines running Debian Linux 3.0. A 100 Mbit, switched Ethernet with a 63 μsec round-trip time connected the client and server. Each machine had a 100 Mbit Tulip Ethernet card. The server had 512 Mbytes of memory. The client had 896 Mbytes of memory.

We configured both REX and SSH to use equivalently-sized cryptographic keys. For authentication and forward secrecy, SFS uses the Rabin-Williams cryptosystem [28] with 1,024-bit keys. SSH uses RSA [21] with 1,024-bit keys for authentication and Diffie-Hellman [3] with 768-bit ephemeral keys for forward secrecy.

We configured SSH and SFS to use the ARC4 [8] cipher for confidentiality. For integrity, SFS uses a SHA-1-based message authentication code while SSH uses HMAC-SHA-1 [4, 11].

4.2.1 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 a lower latency for subsequent logins because of connection caching.

Protocol	Latency
SSH	0.25 sec
REX (initial connection)	0.20 sec
REX (subsequent connection)	0.04 sec

Table 1: Latency of SSH and REX logins

Table 1 reports the average latency of 100 remote logins in wall clock time. In each connection, we remotely log in and are then immediately logged out. We set the shell of the user to `/bin/true`. The user's home directory is on a local file system. For both REX and SSH we disable all port forwarding.

The SSH measurements varied between 0.24 sec and 0.27 sec. The REX measurements varied between 0.19 sec and 0.26 sec. The REX measurements with caching varied between 0.04 sec and 0.05 sec.

The results demonstrate that an initial REX login performs slightly faster than an SSH login. In both cases, most of the cost is attributable to the computational cost of modular exponentiations. The initial connection in REX requires two concurrent 1,024-bit Rabin-Williams decryptions—one on the client and one on the server—followed by a 1,024-bit Rabin-Williams signature on the client. (All three operations are Chinese remaindered.) An SSH login performs one 768-bit Diffie-Hellman key exchange and two 1,024-bit RSA decryptions. REX's operations are less computationally expensive than SSH's, and thus, as expected, initial REX connections are slightly faster. Cached REX connections require no public key cryptography, and therefore take only 0.04 sec. If SSH were to implement connection caching, presumably it would also be able to achieve low subsequent connect latencies.

4.2.2 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 [23]. NetPipe streams data using a variety of block sizes to find peak throughput.

Protocol	Throughput	Latency
TCP	89.7 Mbit/sec	63 μsec
SSH	54.0 Mbit/sec	154 μsec
REX	58.5 Mbit/sec	262 μsec

Table 2: Throughput and round-trip 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 89.7 Mbit/sec. Next, we measure the throughput of a forwarded port over an established SSH and REX connection. Table 2 shows that REX has a slightly better throughput than SSH.

We attribute the additional latency of ports forwarded with REX to three RPCs necessary to pass a file descriptor before port forwarding can begin. While latency of ports forwarded with REX is greater than with SSH, the sub-millisecond latency of REX is still very small in absolute terms. Round-trip latency to a well-placed machine on the Internet is an order of magnitude greater than the latency of a port forwarded with REX. Thus, the extra security and flexibility gained from modularization of file descriptor passing do not come at significant cost.

[In the final version of this paper, we intend to show why the latency does not have a noticeable effect on end-to-end measurements such as rsync over SSH versus rsync over

REX. We expect applications such as rsync, which last much longer than the network round-trip times involved, to overshadow this latency. For interactive applications, users would not notice sub-millisecond delays.]

5 Related Work

Several tools exist for secure remote login. This section focuses primarily on the remote login protocol SSH. We also describe other remote execution implementations. This section concludes with a discussion of user agents and connection caching.

5.1 SSH

SSH [32] is the de-facto standard for secure remote execution and login. SSH is decentralized—one only needs local superuser privileges to run the SSH server daemon. One does not need to obtain server certificates or otherwise register with any sort of realm administrator before a person can connect to the SSH server. SSH also offers several modes of user authentication. For example, it has optional support for Kerberos [25], allowing users password-free login to servers plus ticket and AFS [7] token forwarding.

SSH was the main inspiration for REX, as we needed an SSH-like tool that could work with SFS. Although we could have extended SSH for this purpose, we built REX from scratch for two reasons. First, as typically configured, SSH servers read files in users' home directories during user authentication (e.g., `authorized_keys`). This policy is incompatible with our goal of integrating remote login with a secure file system, as the server would generally not have permission to read users' files before those users are authenticated.

The second benefit of building REX from scratch is that by designing a new protocol we could exploit file descriptor passing to simplify the implementation. The REX system is split into separate programs that communicate using RPC and file descriptor passing. Most of the programs that make up REX have fewer than 1,000 lines of code (not counting the general-purpose crypto and RPC libraries we use). Many of the functions built into SSH are instead provided by small external utilities with REX. We could not have taken this approach had we needed to maintain compatibility with SSH.

OpenSSH [17], a popular implementation of the SSH protocol, has recently embraced privilege separation to reduce the number of places where programming errors can cause catastrophic security vulnerabilities [20]. By reducing the amount of trusted code, a programming error is less likely to yield escalated privileges (e.g., root) to an attacker. Privilege separation builds upon the principle of least privilege, which states that a program should enjoy the least privilege necessary to complete its task [22].

We believe that many of the ideas in REX are applicable to SSH and other remote login tools, and hope that SSH and REX will increasingly resemble each other. For example, as part of the privilege separation code in the latest version of OpenSSH, the OpenSSH server internally uses file descriptor passing to handle pseudo-terminals. Even though file descriptor passing is part of the source code, it is not part of the protocol. Generalizing the idea cleanly, though, so that file descriptor passing could be used in other places, would require modification to the SSH protocol, which we hope people will consider for future revisions.

The main difference between REX and SSH is the use of file descriptor passing to provide privilege separation and to extend the program's functions. REX also provides connection caching to improve performance, selective signing to help users who use machines is several administrative domains, and SRP to allow users to both retrieve their private key from the authentication server and to authenticate the host to which they are connecting. SRP sidesteps the need to deal to potential man-in-the-middle attacks.

5.2 Kerberos

Kerberos [25] is a centralized authentication system which comes with remote login and execution utilities. It provides a unified way of naming, authenticating, and authorizing principals. In Kerberos, however, users must be organized into realms. Joining an existing realm (i.e., setting up a server) requires permission from and coordination with that realm's administrator. In part because Kerberos is based on shared-secret cryptography, creating a new realm is not a simple task and still requires administrative permission to inter-operate with existing realms.

Kerberized remote login is based on this centralized architecture and must use a trusted third party (the KDC) for mutual client and server authentication. While REX and SFS support third-party authentication, it is not required. Clients and servers can authenticate each other in whatever manner is desired.

The AFS [7] file system uses Kerberos to authenticate users. The Kerberos remote login tools forward AFS tokens and authenticate users to the file system before logging in the user. REX and the SFS agent provide similar support for the SFS file system.

The GSS-API is a generic interface for client-server authentication [13]. Because REX was directly designed for interoperability with SFS, the GSS-API was not necessary. REX's design does not preclude the use of GSS-API in the future, however.

5.3 Secure rlogin

Before SSH, researchers explored other options for secure remote login [10, 27]. Kim et al. [10] implemented a secure *rlogin* environment using a security layer beneath TCP. The system defended against vulnerabilities created by host name-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 [9].

5.4 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. SSH, for example, has agents capable of authenticating users to servers. For other tasks such as server authentication, however, it 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 by simply 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. This design makes it impossible to implement selective signing in SSH agents.

Recently, the security architecture for the Plan 9 system has been redesigned [2]. The new Plan 9 architecture has an agent, *factotum*, which is similar to an SSH and SFS agent, but is implemented as a file server.

The Taos operating system [12, 29] and the Echo file system [1] also have a notion of an authentication agent. Unlike SFS, their agents refer to a component of the operating system rather than a user-controlled process.

5.5 File descriptor passing

We note that an alternative to file descriptor passing would be file namespace passing, as is done in Plan 9 [18]. 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 reason tools like SSH have developed so many different ad hoc mechanisms for handling different types of resource.

5.6 Connection caching

The idea of session resumption is available in other systems such as SSL [5]. *Fsh* [6], a wrapper around SSH that attempts to reuse its encrypted channel to execute subsequent commands, demonstrates the interest in such a feature.

6 Conclusions

REX provides secure remote login and execution in the tradition of *rsh* and SSH. REX departs from the design of its predecessors by applying and extending two existing ideas—file descriptor passing and a user agent.

Local file descriptor passing enables privileged code separation. The current REX implementation requires roughly 800 lines of code run with superuser privileges. REX's ability to emulate file descriptor passing over the network allows users to build new features outside of the core software. Many of the extra functions that are part of SSH are written as separate modules in REX. File descriptor passing simplifies REX's design and is a powerful primitive for building a remote execution facility.

The SFS user agent allows REX to provide a consistent computing environment across machines. REX makes this "secure computing environment" available on other machines by forwarding the agent to remote login sessions. Remote and local programs then have access to the same private keys, server nicknames, etc.

The REX architecture provides a convenient, extensible interface to the user and developer without compromising security or efficiency. Rather, REX provides enhanced security by reducing the amount of privileged code. The increased extensibility comes at no extra cost. REX is as efficient as existing tools and can even do better by caching connection state.

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

References

[1] 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.

[2] 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.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

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

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] J. Linn. Generic security service application program interface version 2, update 1. RFC 2743, Network Working Group, January 2000.

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

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

[16] 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. ACM.

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

[18] 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.

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

[20] Niels Provos. Preventing privilege escalation. Technical Report TR-02-2, University of Michigan, CITI, August 2002. `http://www.citi.umich.edu/u/provos/ssh/privsep.html`.

[21] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

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

[23] 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`.

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

[25] 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.

[26] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley Longman, Inc., 1993.

[27] 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.

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

[29] 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.

[30] 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.

[31] T. Ylönen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH transport layer protocol. Internet draft (draft-ietf-secsh-transport-14.txt), Network Working Group, March 2002. Work in progress.

[32] 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.

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