Preserving Integrity in Remote File Location and Retrieval

Trent Jaeger [*]
jaegert@eecs.umich.edu
Software Systems Research Lab
EECS Department
University of Michigan
Ann Arbor, MI 48105

Aviel D. Rubin
rubin@bellcore.com
Security Research Group
Bellcore
445 South Street
Morristown, NJ 07960

Abstract

We present a service for locating and retrieving files from an untrusted network such that the integrity of the retrieved files can be verified. This service enables groups of people in geographically remote locations to share files using an untrusted network. For example, distribution of an organization's software to all the organization's sites can be accomplished using this service. Distribution of files in an untrusted network is complicated by two issues: (1) location of files and (2) verification of file integrity. `ftp` *and World-wide Web (WWW) services require some user intervention to locate a file, so they cannot be embedded in automated systems. Distributed systems have mechanisms for automated file location and retrieval, but they require trust in all system principals and do not provide an appropriate balance between availability of files and retrieval cost for our applications. Verification of the integrity of a file retrieved from an untrusted network is necessary because the file is subject to malicious modification attacks. Our service provides the capability to automatically locate, retrieve, and verify files specified by a client using a single trusted principal. We demonstrate our service by building a system shell that automatically downloads remote software when needed.*

Keywords: Digital signatures, cryptographic digests, remote procedure calls, wide-area network file location, trusted authorities, C-shells.

1 Introduction

We present a service that automatically locates and retrieves files from an untrusted network and verifies that the file retrieved is the file requested. This service enables users to share files distributed over the Internet. For example, a user may want to retrieve a technical document with a specific document number from a remote server in a geographically distributed organization. A user can use this service to automatically retrieve such a document, even if the user does not know the identity of the machine that stores the file. In addition, the service provides a mechanism that verifies the authenticity of the document. The service solves two major problems: (1) ensuring authenticity of information obtained from an untrusted network and (2) locating distributed information.

Many users already use tools such as `ftp` to download and run software from the Internet, but this is a very dangerous practice. For example, a malicious attacker may replace software, either in transit or on a compromised machine, with malicious software. Since the malicious software is run with the user's access rights, this software can: (1) access the user's private data; (2) delete the user's data; (3) access system information, such as a password file; and (4) tie up system resources. Even worse, the malicious software may contain a virus or a Trojan horse. Bellcore's Trusted Software Integrity (Betsi) system [14] enables users to manually verify the integrity of a file obtained from the Internet. Betsi requires user involvement to verify that the certificate is for the file requested and to compare cryptographic digests and digital signatures to those in the certificate, so it cannot be integrated with our service in its current form.

A second problem with file retrieval from the Internet is that it is difficult to locate a specific file

[*] This work was done while this author was a summer intern at Bellcore.

without knowledge of its specific location or significant user involvement. To retrieve a file using `ftp`, the user must know the name of the ftp site in addition to the name of the file. Also, current web searching programs aid the user in performing keyword searches of the Internet, but require users to guide the search directly. Therefore, these systems are prone to user errors and cannot be embedded within an application. Distributed file systems and shared virtual memory systems provide mechanisms for automated location of files, but they have the following limitations: (1) they do not provide the proper trade-off between pushing files to clients and pulling files from servers and (2) they require that too many system principals are trusted.

The goal of our service is to automate the file location, retrieval, and authentication tasks of the client. Using our service, a client can specify the file to be retrieved and expect the service to return a copy that meets those specifications, if one is available. Files can be located efficiently without requiring trust in the locating services. In fact, only one trusted principal is required for the protocol to succeed. We demonstrate the types of tools that can be constructed from the service by building a system shell that automatically locates and downloads software from remote machines.

In the next section, we define the problem. In section 3, we present the service architecture. In section 4, we describe the implementation of the service. In section 5, we present an application of the service.

2 Problem Statement

Our service must solve two basic problems: (1) it must be able to locate and retrieve the file requested by the client and (2) it must be able to verify that the file retrieved satisfies the client's request. In this section, we formally define these two problems.

2.1 File Location and Retrieval

Before defining the file location and retrieval problem we define the following concepts:

- **Definition 1**: A *file identifier* is a tuple that includes the identifying characteristics of a file, such as its name, its author, its date, etc [1].
- **Definition 2**: A *file* is a stream of bits referenced by a file identifier.
- **Definition 3**: A *client* is a principal that requests a file using a file identifier.
- **Definition 4**: A *distribution server* is a principal that can process a file identifier and determine if it possesses a file that maps to the identifier.

When the client wants to retrieve a file, it specifies a file identifier. The file location problem is for a client to identify a distribution server that can map the file identifier to a file that it possesses. The file retrieval problem is for the client to obtain an authentic version of that file from a distribution server.

Current file location mechanisms can be divided into two categories: (1) pull; and (2) push. Pull systems, such as V [4, 5] and shared virtual memory systems like Ivy [9] and Emerald [6], generate file location information on demand and cache the information locally. Location of new files requires a broadcast, so it can be expensive. V defines domain-wide caches, called *liaison servers*, for storing location information generated by all clients in a domain. Liaison servers reduce the number of broadcasts that are necessary since a file that has been located once is stored by at least one liaison server.

In push systems, such as Grapevine [3], DEC Global Name Service [7], and Ameoba [17], servers generate information about remote files and distribute that information to clients. The push model makes retrieval more efficient because the information about the location of each remote file is closer to the client. However, the cost of forwarding change messages can be high. In Ameoba, only 'published' files are made available to the network, so only the location of these files needs to be known by remote machines. However, Ameoba requires that clients run server agent processes on their machines to locate published files. We question the feasibility of this approach because: (1) it is expensive to publish information to every machine in the Internet and (2) allowing processes to be triggered by the actions of foreign machines presents a potential security vulnerability, particularly when these processes may affect the way that the local machine executes future processes. However, we expect that people responsible for distribution servers will know what files they want to offer, such as in the WWW where users make file URL's available, so the push model is preferred.

Another problem with the systems described above is the amount of trust that is required for their file location mechanisms to succeed. All these systems assume that the distribution server has a valid copy of the requested file and that the location server is trusted. For example, the V system requires that its

[1] A detailed definition of the file identifier used is provided in Section 4.2.

managers (i.e., distribution servers) sign all messages. However, even if the data is signed, the compromise of a distribution server may result in the use of invalid applications or data. Also, V assumes that its liaison servers are trusted and the integrity of their communications with clients is preserved. Trust in the location server and its communications breaks down if it is subject to compromise or communicates over an untrusted network. We expect that distribution servers and location servers connected to the Internet will be attacked, so applications, replicated data, or communications delivered from these machines cannot be trusted.

Current systems assume that once a file is found, it is trivial to retrieve it. This assumption is not true if the file resides in an untrusted network. Malicious attackers can modify the file in transit. Resending a file may not be sufficient because the file may have been corrupted when a distribution server was compromised. The result is that a client may have to retrieve a file from a distribution server multiple times or from another distribution server.

2.2 File Authentication

The file authentication problem is to prove that the file retrieved meets the requirements of a client's file request. For example, a client may request a software package named X. However, if a distribution server supplies a file named X, that is not sufficient to verify to the client that the package is indeed software package X. The distribution server could have just renamed another file X.

More formally, the file authentication problem is for a client to verify that the following statements about a file *f* are true:

- **Statement 1**: The identity of a file *f* matches the identity in the file identifier provided by the client.
- **Statement 2**: The author of file *f* matches one of the authors in the file identifier for *f*.
- **Statement 3**: A cryptographic digest of file *f* matches a cryptographic digest of a file whose identity matches *f*.
- **Statement 4**: The expiration date of the cryptographic digest, identity, and author of a file *f* has not passed.

Statement 1 says that the retrieved file's identity matches that of the request. Statement 2 says that the file is authored by an author approved by the client. Statement 3 says that the retrieved file's contents correspond to the file contents expected. Statement 4 says that the file authentication information is current.

We assume that a client believes any statement made and digitally signed by a trusted authority. Therefore, a client can compare the file retrieved to authentication information signed by a trusted authority to determine if a file is authentic. This is the protocol implemented by the Betsi system [14].

In Betsi, a trusted certification authority (CA) creates certificates that associate a registered author with a file identifier and a cryptographic digest of the file. Betsi's author registration protocol prevents an attacker from masquerading as another registered author. These certificates are digitally signed by the CA to ensure their authenticity. Therefore, a user can verify that a file is the one specified in the certificate and that its integrity is preserved. The following protocol is used to verify file authenticity. First, a user compares the file identifier and author in the certificate to the file identifier and author expected by the user. Next, the user computes a cryptographic digest of the file (using a trusted hash program, such as MD5 [12]) and compares this digest to the cryptographic digest in the certificate to verify the file's integrity. If the user trusts the CA and the two comparisons succeed, then the file satisfies the Betsi verification protocol. The requirement for the file to be current is implemented in Betsi via a certificate revocation list.

The main limitation with the Betsi approach is that the user must validate the information in the certificate. This task is arduous and error-prone: (1) users may not know the criteria for accepting a file; (2) users may misinterpret the certificate; and (4) users may become inured to the process and accept files without checking them. However, it is possible to automatically authenticate files using the criteria for authentication given above.

3 Architecture

Our service's architecture is based on the following assumptions. First, each client can securely obtain a copy of the CA's public key. An off-line mechanism can be used to distribute this key. Second, we assume that an attacker cannot certify files using another author's identity. Betsi uses an off-line mechanism to verify an author's identity during registration of an author's public key. We do not expect that Betsi's mechanism will be used for this application, but we require that some off-line verification is performed.

Third, we assume that trusted software for generating cryptographic digests and for verifying digital signatures are available at each client. Finally, denial-of-service attacks are generally easy to detect and hard to prevent, so we assume that they can be detected by the system and are repaired off-line.

The trust model includes the following principals:

- **Clients**: Principals that request files
- **Distribution Servers**: Principals that store and distribute files
- **Authors**: Principals responsible for a file (e.g., the system administrator of a server)
- **Certification authorities (CA)**: Principals trusted by clients to certify authors' files
- **Location server**: Principals that map file identifiers to distribution servers

CA's certify the authenticity of each file. This contrasts with the PEM [2] model where a CA would certify the public key of each author, and authors would certify files directly. The advantages of CA's certifying files are that: (1) the certification date of the file can be trusted; and (2) certification using previously revoked public keys is prevented. The authenticity of some documents, such as legal documents, may depend on the date that they were certified. Since the CA creates the certificate containing the certification date, this date can be trusted. Also, the CA generates all certificates, so the CA can prevent certification of files signed using a revoked key. In addition, our CA can also support implementation of certificates based on the PEM model, where appropriate.

The client need only trust a single CA in our architecture. At present, we assume that each organization will have one CA. If a group of organizations wants to share information, a web of trust between the CA's of those organizations can be created using the mechanism used for PEM [2] or PGP [18] [2].

Clients need not trust location servers nor distribution servers. If a distribution server delivers an incorrect or tampered file, the client recognizes it. Any change to the certificate is detected by signature verification, and any change to the file is detected by the digest comparison. If a location server is compromised, the client can use another location server. At worst, compromised distribution servers and location servers can cause a denial of service.

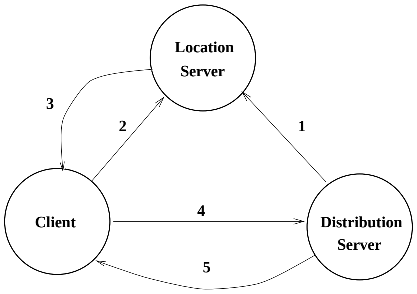

Figure 1: **Service Process:** (1) distribution server publishes a file; (2) client requests a file's location; (3) location server returns a set of distribution servers for a file; (4) client requests a file from a distribution server; and (5) distribution server returns a file and its certificate for authentication.

The service architecture implements the process shown in Figure 1. In step 1, distribution servers announce to location servers that they possess a specific file. In step 2, a client requests the locations of a file from a location server. In step 3, the location server returns a list of distribution servers for the file to the client. In step 4, the client chooses a distribution server and requests the specified file and its certificate. In step 5, a distribution server returns a copy of the file and its certificate to the client. Steps 4 and 5 are repeated until the client receives an authentic version of the file.

The location and retrieval steps differ from distributed operating system location services in a few key ways. Unlike V, a push model is used by distribution servers, so all distributed files are already cached in the location servers. Therefore, broadcasts to distribution servers for files are not necessary. Unlike Ameoba, no server agent processes are required on the client's machine to find a file. Thus, the need for a client to run processes for a distribution server is removed. We also expect that other types of files, such as documents, may be retrieved using our service, so the number of server agents required would be higher than Ameoba's designers anticipated.

Another difference between our architecture and that of V and Ameoba is the number of trusted principals. Clients in both Ameoba and V trust all the location and distribution servers. In an untrusted network, one of these machines is bound to be compromised, so this assumption is not valid for the Internet. In the architecture presented here, only the CA is trusted. An implication of having only one trusted principal is that each client is now required to perform verification of retrieved files. Thus, a client must store the public key of a CA and possess trusted cryp-

[2] PGP is a trademark of Philip Zimmermann.

tographic software.

4 Implementation

Our service is implemented as a set of communicating principals distributed in an untrusted network. There are six types of principals in the implementation: (1) CA's; (2) location servers; (3) distribution servers; (4) publishers; (5) authors; and (6) clients. The actions of CA's and authors are assumed to take place off-line, so their implementation is not detailed. Location servers and distribution servers handle requests from clients and maintain permanent state. The publisher is added to perform the task of uploading information about files available on a distribution server's file system to the location server and to the distribution server itself.

The instances of the six principals are distributed in the following manner. At present, we have implemented only one trusted CA. There is no reason that others cannot be added to the implementation, however. There can be an unlimited number of distribution servers, publishers, clients, and location servers in the network. At present, we have implemented only a single location server, however.

The only cryptographic keys that are needed in this implementation are the CA's public/private key pair. Only the CA has access to its private key. We assume that the CA is on a secure machine. Clients are the only principals that need the CA's public key. They store the CA's public key in a read-only file. In the future, the use of secure hardware (e.g., smart cards) should be considered.

In our implementation, all digital signatures are generated using the Digital Signature Algorithm (DSA) [1]. DSA was developed and patented by the National Security Agency (NSA) and is the proposed standard digital signature algorithm of the National Institute of Standards and Technology (NIST). DSA is a public key algorithm whose security is based on the hardness of the discrete logarithm problem. Our choice of DSA may be somewhat controversial given that a large majority of digital signature applications use the RSA [13] algorithm. However, DSA has the advantage that NIST claims that DSA can be exported and that it is royalty-free [3]. On the other hand, the exportation of RSA is tightly controlled and its use requires royalty payments to the patent holders. Since there is no known way to break DSA, it is a viable alternative with fewer restrictions on its use.

The version of DSA used in our implementation is part of the Long Integer Package (LIP) [8] developed by Arjen Lenstra. LIP provides a library of functions for computing with arbitrarily long integers. For example, LIP can generate 1024-bit integer DSA keys and perform the modular exponentiation operations necessary for signature generation and verification. A free version of LIP is available via `ftp` from the site `ox.ac.uk` at `/pub/math/freelip/freelip_1.0.tar.gz`. This version contains all the functionality of the Bellcore-proprietary version of LIP we used, except for the DSA code itself. However, the DSA algorithm is widely published [15], and it can implemented using the basic LIP functions.

The MD5 [12] hash algorithm is used to generate cryptographic digests. MD5 has the following features: (1) it is a *one-way function* because it is thought to be computationally infeasible to derive the input to an MD5 calculation given the output and (2) it is a *collision-free function* because, given an input and an output, it is possible to find another input that hashes to that same output with only a negligible probability. These features and the fact that the code is in the public domain have made MD5 an RFC standard one-way hash function.

Interaction between principals is implemented using the Sun RPC mechanism [16]. Each of the steps shown in Figure 1 correspond to an RPC request and reply. The publisher and client principals make RPC requests to the appropriate server principal as necessary. The server then responds with a result. For example, to locate a file, a client makes an RPC request that includes a file identifier to a location server. The location server returns an array of distribution servers to the client. The location servers and distribution servers are implemented as Internet services. To make these Internet services available, they must be registered. Registration of Internet services is accomplished by adding the appropriate entries to `/etc/inetd.conf`, `/etc/services`, and `/etc/rpc` files. Below, we detail the contents of the RPC requests and the actions taken upon receipt of such requests.

4.1 Initial Conditions

Before detailing the implementation, we identify the initial conditions assumed by the implementation. First, we assume that a publisher wants to make a file accessible to clients. This publisher may or may not be the author (e.g., developer) of the file.

Registered authors certify their files with a CA trusted by those clients who will want to use the file.

[3] It is currently unresolved as to whether a patent by Schnorr covers DSA, however.

This enables clients to verify the contents of the file, its identifying attributes, and its certification date. The CA provides the author with an authentication certificate for the file. We assume that the publisher obtains a copy of the file and its authentication certificate off-line.

Effectively, a file authentication certificate associates an author with a cryptographic digest of a specified file. Digests are used in the place of files because hashes have a small, fixed size (e.g., 128 bits for MD5). Clients can compare a digest of the file to the digest provided in the certificate to verify the integrity of the file. The integrity of the certificate is guaranteed by the signature of the trusted CA.

Other information is needed in the certificate to verify that the requested file was retrieved. Therefore, a certificate has the following fields:

- Identity of CA
- Author Name
- Author's Organization
- Author's E-Mail Address
- Name of the File
- Version Number
- Machine
- Machine ID
- O/S
- O/S version
- Cryptographic Digest
- Expiration Date
- Latest Version?
- Date

The fields in the certificate have the following meanings. The `identity of CA` is used so the client can determine which public key to use to verify the authenticity of the certificate. Author information enables the client to verify the author of the file. The `version number` of the file enables the client to verify that the file is the proper version. The machine and O/S information are used to verify that the file is appropriate for a specific platform. These fields are applicable to software. As described above, the `cryptographic digest` is used to verify the integrity of the file. The expiration date indicates the date when the certificate becomes invalid. The `latest version?` field specifies that the author claims that this file is the latest version at the date the certificate was created. The `date` is the date of certification. The protocol for verifying the authenticity of a file using these certificates is detailed in the File Authentication Section below.

4.2 File Publication

Clients can locate a file on a particular distribution server because it is 'published' to the network. The act of 'publishing' was first used in the Ameoba system [17] as a way to advertise that a particular service resides on a particular server. In Ameoba, a file publication protocol activates server agents on client machines to catch and forward service requests. In contrast, our implementation has a publisher process that uploads file location information to both location servers and distribution servers.

An important consideration in file publication is the specification of a file. A file must be specified in a way that it can be uniquely referenced. A unique file is specified using a file identifier. A file identifier consists of the following fields:

- File Name
- Author Set
- Version Number (optional)
- Machine (required for compiled software only)
- Machine version (required for compiled software only)
- O/S (required for compiled software only)
- O/S version (required for compiled software only)
- Latest Version? (optional)

A client is required to know the execution platform for compiled software. The execution platform can either be entered by the client or stored in a file. Otherwise, these values are not required. If the version is not specified, then it is assumed that the `latest version?` of the file is requested.

Location servers store a map of file identifiers to the set of servers that provide them. Our implementation allows for a distributed model of location servers, similar to the model of liaison servers in the V operating system [5]. Liaison servers update their database as they fulfill client requests. In our implementation, only published files can be retrieved and the location of these files is uploaded to location servers.

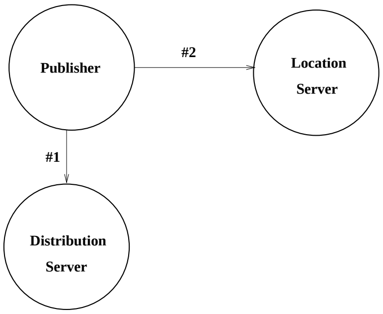

Figure 2: **The file publication protocol:** (1) Publisher uploads a file identifier-to-pathname mapping to distribution server; (2) Publisher uploads a file identifier-to-distribution server mapping to location server.

Distribution servers store a map from a file identifier to the file pathname and the file's certificate pathname on the distribution server's file system. This enables a distribution server to quickly locate a file requested by a client.

The file publication protocol is shown in Figure 2. The protocol steps are as follows:

1. The publisher informs the distribution server to update its file identifier-to-pathname database. Each distribution server database entry has the following fields:
   - File Identifier
   - File Pathname
   - Certificate Pathname
2. The publisher tells a location server to update its file identifier-to-server database. Each location server database entry has the following fields:
   - File Identifier
   - Distribution Server

The results of the file publication protocol are: (1) the location server can list the set of distribution servers that claim to possess a file that matches a file identifier and (2) the distribution server can locate that file and the file's certificate given a file identifier.

In this implementation, it is possible for attackers to publish false files on a location server. This attack will not result in a client accepting a file that is not authentic, but can result in a denial-of-service if the location server database becomes full of false file locations. Attackers can be prevented from publishing false files if the location server also requires a publisher to present a certificate for the file upon publishing. This does not prevent an attacker from publishing legitimate files, however. Off-line verification of published software by system administrators can be used to limit the number of false entries.

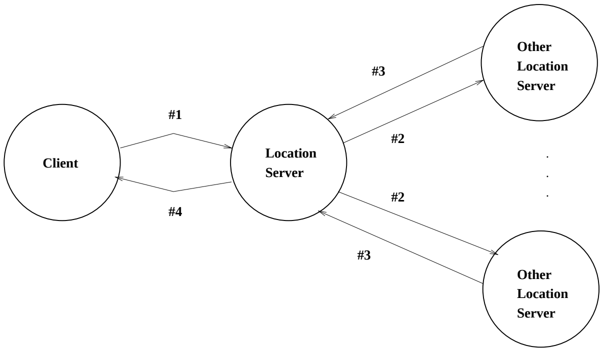

Figure 3: **The file location protocol:** (1) Client sends a file identifier to the client's location server to find the distribution servers that possess a file; (2) Client's location server requests that other location servers find the distribution servers that possess a file (optional); (3) Other location servers return to the client's location server a set of distribution servers that possess a file; (4) Client's location server returns a set of distribution servers that possess the file.

4.3 File Location

The file location protocol provides a client with a set of distribution servers that claim to possess a file that matches a file identifier. If a client already knows a distribution server with the desired file, then the file location protocol is not necessary.

The file location protocol is shown in Figure 3. The steps in the protocol are as follows:

1. A client sends a file identifier to the client's location server. The file identifier is defined in the File Publication Section above.
2. (Optional) The client's location server sends the file identifier to some number of the other location servers. This is only done if the client's location server has no knowledge of any distribution servers that possess the file.
3. (Optional) These other location servers return location structure messages to the client's location server. The location structure messages have the following format:
   - An Array of Distribution Servers
   - The Number of Distribution Servers

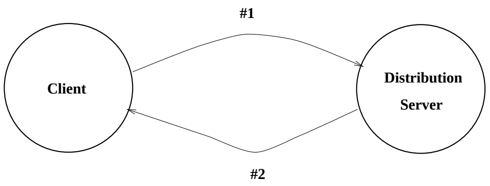

Figure 4: **The file retrieval protocol:** (1) client sends a file identifier to a distribution server; (2) the distribution server returns the corresponding file and its certificate.

4. The client's location server returns a location structure message to the client.

The result is that the client collects a set of distribution servers (possibly empty) that the client's location server claims possess the requested file. We have no requirement that location servers are consistent, so if a location server has no entry that matches a file identifier then it can request the help of other location servers. The client's location server caches any new information it obtains from these other location servers. Therefore, the next time a client requests the same file, the location server can provide the set of distribution servers found previously. We expect in future implementations that location servers will publish new files among one another at regular increments (e.g., daily) or after a threshold number of updates have been made to them. Protocols for this activity are beyond the scope of the paper.

At present, we have implemented only messages 1 and 4 of the file location protocol although we see no reason why the implementation of messages 2 and 3 should be problematic.

The main attack against file location is for an attacker to modify the location server's return messages (the location structure messages) with a false reply. Again this attack will only result in a denial-of-service, so client retries can overcome this attack.

4.4 File Retrieval

The file retrieval protocol uses the set of servers obtained in the file location protocol to retrieve a file and its certificate. The file retrieval protocol is fairly straightforward, but it may need to be rerun if the file authentication fails. Because file authentication can fail, the client treats the file location information as a hint to the possible locations of the file. The file retrieval protocol determines how these hints are used.

The base file retrieval protocol is shown in Figure 4. The client sends the same file identifier it sent to the location server in the file location protocol to a distribution server. The distribution server returns the corresponding file and its certificate. A null file indicates that the distribution server does not possess the file. If a file and its certificate are returned, the file authentication protocol described in the next section is run.

If file authentication fails, then the client must determine what the next action should be. The client has four options: (1) retry the retrieval using the same distribution server; (2) try the retrieval using another distribution server; (3) obtain a new distribution server set from the same location server; and (4) obtain a new distribution server set from another location server. Since we expect relatively few failures, we use a simple, round-robin algorithm to locate a distribution server with an authentic copy of the file. First, each distribution server in the distribution server set are tried in succession. If all the distribution servers fail (the client can specify a limit for the number of failures), another location server is queried for a new set of distribution servers and the algorithm is repeated. If a client detects a failure, it notifies the server's system administrator. Verification of server state and failure repair are performed off-line.

4.5 File Authentication

Once a client receives a file and its certificate, the client needs to verify that the file was the one requested. The file authentication protocol compares the file identifier generated by the client and the file itself to the information in the file's certificate. The result is that the client verifies that the file is authentic or discards the file.

The file authentication protocol consists of two steps: (1) verifying the CA's signature and (2) verifying the information in the certificate. The verification of the CA's signature is necessary to ensure that the certificate has not been modified. Any change to the certificate will cause the digital signature verification process to fail.

Once the CA's signature is verified, the information in the certificate is checked to ensure that the correct file has been retrieved. The file and its identifier are compared to the following components of the certificate:

1. Expiration Date of the Certificate

2. File Name

3. File Author

4. Version Number (optional)
5. Latest Version Number (optional)
6. Machine (optional)
7. Machine Version (optional)
8. O/S (optional)
9. O/S Version (optional)
10. Cryptographic Digest of the File

First, the expiration date of the certificate is checked. The expiration date indicates the time at which a certificate becomes invalid. An author can use this capability to register files until a certain date. Then, the author can either re-certify the file or certify a new version of the file. This is useful for releasing updated versions of documents or software. A detailed example of this is provided in the Applications Section.

Next, the file identifier information (e.g., file name, author, etc.) can be verified. This is accomplished by matching the file identifier information supplied by the client against the values of the file identifier attributes in the certificate. The author field verification is satisfied if the author matches one of the author's in the file identifier's `author set`.

Lastly, the integrity of the file itself is checked by computing the cryptographic digest of the file using MD5 and comparing that value to the cryptographic digest in the certificate. If the digests match, the integrity of the software is assured with a large degree of confidence. This assurance is possible because the CA's signature guarantees that the validity of the digest and the probability that MD5 computes the same digest for two distinct inputs is negligible.

Even if the hashes match, it is possible that the author may have certified malicious software in the first place. In this case, the author is directly linked to the software, so any malicious actions by the software can be attributed to the author.

5 We Sell C-Shells

We have implemented a shell interpreter that obtains remote software on demand and ensures that the retrieved software is the software requested before executing it. This shell, called the *Secure sofTware RE-Trieval SHell* or *stretsh*, automatically retrieves and authenticates software from the network, such that it appears that files made available on the network are in the normal execution path of the client. Therefore, no user involvement is necessary to execute applications whose software resides on remote file systems. Also, `stretsh` enforces software version control by automatically downloading new versions of software when the previous version's certificates expire.

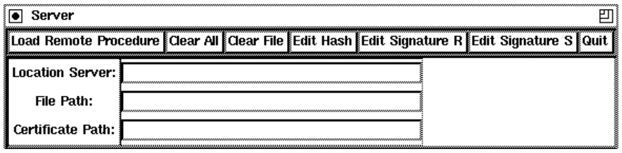

Figure 5: A form for publishing files to a location server.

`Stretsh` is a shell language developed for Unix [4]. Although the functionality of this shell language is not Unix-specific, our prototype implementation is specific to Unix. `Stretsh` extends the `csh` scripting language by placing a wrapper around command execution that locates a command before it is executed. Unix-specific utilities are used to locate commands.

As an example, consider an employee accounts application in a geographically distributed organization. This application has a main program that provides access to a set of utility software for timesheets, expense reports, account management, etc. `Stretsh` scripts can download these utilities and verify their authenticity automatically. In addition, when new versions of utilities are published these can be downloaded as well.

System administrators announce new software is available using the file publication protocol described above. A form is defined for system administrators to enter the publication information. This form is created using Tcl/Tk [11] and shown in Figure 5. The `location server` field in the form specifies the domain name of the location server. The `file path` field indicates the file name and the location of the file. The `certificate path` field indicates the location of the file's certificate. The file identifier attribute values are obtained from the certificate. The command `Load Remote Procedure` initiates execution of the file publication protocol.

Once a software package is published, `stretsh` can retrieve it. `Stretsh` retrieves software requested by its shell programs if it is not already in the execution path of the user. When a program is to be executed, `stretsh` first checks to determine if the program is in the local execution path using `which` [5]. The pro-

[4] Unix is a registered trademark of the Unix Open Foundation, Inc.

[5] `Which` is a Unix command that locates a command if it is one of a set of specified directories.

totype stretsh implementation only checks the first entry in the command line, but all executable software could be checked during the construction of the execve command. If the software is present, then the program is executed in the normal fashion. If not, stretsh attempts to retrieve the file. First, stretsh builds a file identifier for the file. The file identifier attribute values are obtained in the following manner:

- **File Name**: Name of software
- **Author Set**: From .trusted_authors
- **Version Number**: In <script_name>.pre or Null
- **Machine**: From .platform
- **Machine version**: From .platform
- **O/S**: From .platform
- **O/S version**: From .platform
- **Latest Version?**: True unless a version is specified in <script_name>.pre

The set of authors trusted to write software that can be retrieved are listed in a .trusted_authors file maintained by the system administrators. Thus, the name of a software package must be unique to an author. Author names are suggested by the authors, but approved by the CA. The platform information is stored in a .platform file for each machine (set at login time). Both the .trusted_authors and .platform files reside on the local machine. Unless a version specification is provided for the software in <script_name>.pre, the latest version of the software is obtained by stretsh. The location server then returns a set of distribution servers to stretsh. Stretsh uses the same file identifier to retrieve a copy of the software and its certificate from a distribution server. Software and their certificates are copied to the directories indicated by the environment variables $STRETSH_DIR and $STRETSH_CERT, respectively. The file authentication protocol is then executed on these files as described above. If file authentication is successful, the software can be executed.

Consider the employee accounts application again. If a new expense report program, called expenser, is added to the application script, it can be automatically downloaded by stretsh for a user. The set of trusted authors would include a set of people authorized to certify the organization's software. In this case, the latest version is preferred, so no version information is required in <script_name>.pre. The rest of the file identifier information is obtained from .platform. When request, stretsh locates and downloads a copy of the new expenser software and its certificate to the appropriate directories. The file authentication is then performed on this software. If the authentication succeeds, expenser is executed.

In addition, stretsh can support software version control. Recall that file certificates have expiration dates. An author can use these expiration dates to specify when a software package becomes obsolete. When the certificate expires, stretsh retrieves the new version of the software. In general, the author should specify the earliest date for which the next version of the software may appear. In our example, if a new version of the expenser software is released no more often than once per month, then the expiration date should be set at one month after the certification date. If no succeeding version is released by that date, a new certificate can be created to extend the expiration date of the current version.

At present, we have just begun the implementation of the version control mechanism, so below we describe its design. We define a version controller principal that maintains information about what software has been downloaded, the location of that software and its certificate, and when its certificate expires. When the version controller detects that a certificate has expired it simply deletes the expired software and certificate. When the software is requested stretsh automatically retrieves the current version (may be the same version which has been recertified). If an old version is required, then it can be re-downloaded by explicitly requesting that version. The version controller should be run once per day since the expiration granularity is one day.

6 Conclusions and Future Work

We have presented a service that automatically locates and retrieves files from an untrusted network and verifies whether the file retrieved is the file requested. The service relies on only one trusted principal, called a certification authority (CA), to enable a client to authenticate a file. CA's generate and sign certificates that associate an author with a file and a cryptographic digest of the file. Using these certificates, a client can verify that a file is the one requested and that it was certified by a specific author. Therefore, malicious attackers cannot spoof a client into accepting a modified version of the file, and all content and behavior of an authenticated file can be traced to its author.

Automated location is possible because all remote files are published with location servers available to clients. Clients can query a location server for the identity of the distribution servers that possess a copy of a file. Clients then retrieve a copy of the file from one of the distribution servers. Clients control the location and retrieval protocols, which enables a client to retrieve a file without having to trust either the location servers or the distribution servers.

We demonstrate the application of this service with a new system shell that can utilize remote software. The system shell, called `stretsh`, enables software on an untrusted network to be made available to applications. Application scripts written for `stretsh` can download software published in the location servers automatically for the client. This enables clients to run applications whose software resides on multiple file systems.

In the future, we plan to add features that make the service more useful and reduce the degree to which client's must trust the CA. Administration of files downloaded from the network may become a problem. If other applications use a different software package that performs similar functions, then the client's file system will become a mess. Determination of whether to permanently store retrieved files and how to organize related files would help. Another issue is that the service's security is based on the trust in the CA's. However, it would be preferable to not require trust in any system principal. Merkle [10] defines a mechanism by which malicious action on the part of a CA can be detected by outside observers. With this enhancement, the requirement for complete trust in the CA is eliminated.

Acknowledgements

We thank the anonymous referees for their many valuable comments.

References

[1] NIST FIPS PUB XX, Digital Signature Standard, February 1993. National Institute of Standards and Technology, U.S. Department of Commerce DRAFT.

[2] D. Balenson. Privacy enhancement for Internet electronic mail: Part III: algorithms, modes, and identifiers, February 1993. Internet RFC 1423.

[3] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of ACM*, 25(4):260–274, April 1982.

[4] D. R. Cheriton. The V distributed system. *Communications of ACM*, 31(3):314–333, March 1988.

[5] D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault-tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[6] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[7] B. Lampson. Designing a global name service. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 1–10, 1986.

[8] A. K. Lenstra. Documentation of LIP, March 1995. Bellcore TM-24936. Available via anonymous `ftp` from `flash.bellcore.com` (soon `ftp.bellcore.com`).

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[10] R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 122–133, 1980.

[11] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[12] R. Rivest. The MD5 message digest algorithm, April 1992. Internet RFC 1321.

[13] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[14] A. Rubin. Trusted distribution of software over the Internet. In *Proc. Symposium on Network and Distributed System Security*, 1995.

[15] B. Schneier. *Applied Cryptography.* Wiley & Sons, 1994.

[16] R. Srinivasan. RPC: Remote procedure call protocol specification version 2, August 1995. Internet RFC 1831.

[17] A. S. Tannenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Russom. Experiences with the Ameoba distributed operating system. *Communications of ACM*, 33(12):46–63, December 1990.

[18] P. Zimmermann. PGP user's guide. Distributed by the Massachusetts Institute of Technology, May 1994.