Design and Implementation of a Distributed Authenticated Dictionary and its Applications

Michael T. Goodrich∗  James Lentini†  Michael Shin†
Roberto Tamassia†  Robert Cohen‡

Abstract

We describe the design and implementation of a software architecture that allows for the wide-scale deployment of an authenticated dictionary. Our system, which we call the Secure Transaction Management System (STMS), can be configured to use any authenticated dictionary data structure complying with our interfaces. We present applications of STMS to certificate revocation, document integrity, digital rights management and SSH authentication. We also report on the runtime performance of STMS in various deployment scenarios.

Keywords  authenticated dictionaries, certificate revocation, SSH, secure web services

1 Introduction

The trustworthiness of information is essential in many applications. Indeed, there are several contexts where a violation of the integrity of information could have significant adverse consequences, including digital certificate revocation, financial data reporting, system access control, and wireless device authentication. We are therefore interested in the construction of systems that can provide data integrity services over a large network, such as the Internet.

There is a major bottleneck to building an online trusted data repository, however. The trust essential to the authentication process often times rests with a single entity, such as a certificate authority, financial institution, system administrator, or authentication office. Thus, in order for critical information to be communicated to the entities who need it most, the trusted entity must disseminate it in an authenticated way. However, requiring the trusted authority to answer all client queries online is often infeasible or inefficient. An authority is a trusted source of information, but is not necessarily a large-scale query processor for hundreds or millions of data clients. Also, if the trusted authority is placed online, it could become the target of a denial-of-service attack. Therefore, we are interested in information authentication schemes that adhere to a simple philosophy:

Distribute computations, conserve trust.

That is, we should provide a large number of online locations, called responders, that store a replica of the repository and answer critical queries on the content of the repository. Additionally, we should let the trust in the answers given by the responders be derived entirely from the information source. That is, we view the responders as untrusted parties that are nevertheless able to provide authentication services on behalf of the trusted source. Such a framework is known in the cryptography literature as an authenticated dictionary.

∗Department of Information and Computer Science, University of California, Irvine, goodrich@acm.org.
†Department of Computer Science, Brown University, \{jflentin,mys,rt\}@cs.brown.edu.
‡Univ. of Massachusetts, Boston, rfc@cs.umb.edu.
The theoretical foundations of authenticated dictionaries and related authenticated data structures are studied in [2, 4, 7, 8, 14, 9, 10, 12, 15, 17, 18]. Authenticated dictionaries are related to research in distributed computing (e.g., data replication in a network [4, 16]), data structure design (e.g., program checking [6, 21] and memory checking [5, 11]), and cryptography (e.g., incremental cryptography [3]).

In this paper, we describe the architecture and implementation of a distributed system realizing an authenticated dictionary, called the Secure Transaction Management System (STMS). We also provide an empirical analysis of the performance of STMS in various deployment scenarios and describe various applications of STMS, some of which have been fully implemented.

STMS has significant performance and security advantages over alternate systems, which are based on the following approaches:

- **Exposed source**: the source is online and answers queries from clients signing each response;
- **Repository forwarding**: the source distributes to responders updates to the repository and a signed hash of the repository; a responders forwards to the client the entire repository plus the hash signed by the source as a proof of the answer (the Certificate Revocation List approach);
- **Trusted responder**: the source distributes to the responders updates to the repository; a responder answers a client query with a signed response (the OCSP approach).

We also show how our system can integrate smoothly with a Web services model, such as Microsoft’s .NET framework, using a type of XML signature. Indeed, we describe several Web-based applications of STMS that have been implemented, including end-to-end integrity of Web content, certificate revocation, and SSH host authentication.

The rest of this paper is organized as follows. In Section 3, we describe the software architecture of STMS and discuss its implementation. Experimental results on the practical performance of STMS in various deployment scenarios are presented and analyzed in Section 4. Section 5 presents jssh, a shell client that uses STMS for secure and efficient server authentication. In Section 6, we overview several other fully implemented applications of STMS.

## 2 STMS Trust Model

As outlined in Section 1, STMS provides responders that give responses that are as trusted as if they came from a trusted source. In this section, we outline the STMS trust model and introduce concepts that will be used throughout the rest of the paper. We use two scenarios to demonstrate the relationships between the major STMS subsystems: (i) the source updating the responders and (ii) a client querying a responder.

### 2.1 Updating the Responders

An interaction diagram demonstrating the process for source updates to propagate to the responders is shown in Figure 1.

Step 1 has the source polling the data source for updates. To prevent denial of service attacks, the source does not accept direct updates. Instead, it the source polls data sources for updates, then reads the updates and applies them to its internal data structure. STMS uses the concept of a **time quantum** to indicate the time interval between updates being sent from the source to the responders. Step 2 occurs when the time quantum ends. Before distributing updates, the source calculates and digitally signs the **basis** (Step 3). The basis is a fingerprint of the entire data set stored at the source. The signed basis is used to provide trust in every query during the following time quantum. For details on how the basis is constructed, see [12, 13].

Step 4 is the distribution of updates and the basis to the responders. The repository uses the updates (Step 5) to update its state to match the source. The responder is now ready to accept
queries for this time quantum.

2.2 Querying a Responder

An interaction diagram demonstrating the process for querying a responder is given in figure 2.

In Step 1, a user makes a query for a particular key $x$. The responder returns the following:

- the response,
- the signed basis stored at the responder, and
- a proof of the response against the basis. The proof is a value that when combined with the query key allows the user to validate the response against the basis.

The validation algorithm (Step 2) is much like validating a digital signature: combine the query value with the proof and compare the resulting value with the basis. The trust in the response is derived from the fact that the basis is signed by a trusted party, the source. This technique can be used to validate positive and negative responses, depending on the underlying authenticated data structure. Examples of possible data structures can be found in [2, 12, 13].

3 Architecture of STMS

In this section, we will describe the software architecture of the Secure Transaction Management System (STMS).
STMS involves the interaction of four primary parties: a publisher of data, a trusted source, untrusted responders, and clients. The source maintains a set $S$ of key-value pairs that evolves over time. A publisher is a trusted entity who is given the authority to insert and delete elements in the set $S$. A responder maintains a copy of set $S$ and answers queries from clients of the form “is element $e$ in set $S$?” and “what is the value associated with element $e$?”. Figure 3 shows the high-level architecture of STMS.

The source and responder components have been implemented in Java. Various clients and publisher components have been implemented in Java, C++ and C#. Also, we have developed toolkits for building Java and C# clients.
The rest of this section is organized as follows. Section 3.1 describes the software architecture of the data structures used by the source and repositories. Section 3.2 describes the software surrounding the data structure. Section 3.3 will describe the core subsystems used by STMS to allow publishers and clients to modify and query the data set, respectively.

3.1 Authenticated Dictionaries and Maps

Previous implementations of authenticated dictionaries [2, 12, 13] simply maintain a set $S$ of elements and support containment queries in the form, “is element $x$ contained in $S$”. We extend the interfaces presented in [13] to allow for queries of the form, “what is the value associated with key $x$”. We will refer to this extended authenticated dictionary as an authenticated map as it maintains mappings from keys to values. The previous version which supports only membership queries will continue to be called an authenticated dictionary.

The interfaces that describe the authenticated map are: AuthenticatedMap with its subinterfaces SourceAuthenticatedMap and ResponderAuthenticatedMap, AuthenticResponse, Update, and Basis. Through these interfaces, there are two main operations which can be performed on authenticated maps. They are outlined in the following sections.

Queries

Two types of queries can be performed on an AuthenticatedMap. The relevant methods are

- AuthenticResponse containsKey(Object key),
- AuthenticResponse get(Object key), and
- Basis getBasis()

The first is a simple membership query. The second corresponds to requesting the value associated with the specified key. It is important to note that, depending on the underlying algorithm used to realize the authenticated map, both methods may return the same AuthenticResponse since the value may be required to verify the membership query. Both methods are included in order to keep the AuthenticatedMap interface as consistent as possible with the java.util.HashMap interface. The Basis corresponds to a statement signed by the source against which all authentic responses can be checked against.

On an instance of AuthenticResponse, a client can call the methods containsKey, getKey, getValue to extract the appropriate information. In order to trust these methods, a client must verify that the signature on the instance of Basis is valid and call the method validatesAgainst(Basis) on the instance of AuthenticResponse.

Modifications and Updates

Modifications can be made at the source by calling the following methods:

- Update put(Object key, Object value) and
- Update remove(Object key)

Each of these methods returns an instance of Update which is used by the ResponderAuthenticatedMap to stay consistent with the SourceAuthenticatedMap. The means by which these updates are distributed to the responders will be discussed in Section 3.4.

The SourceAuthenticatedMap interface also defines a method called getInitializationData which returns an object of type AuthenticatedMapInitialization. This object is used as a parameter to the method initialize on the side of the ResponderAuthenticatedMap. After the initialize method is called, the source and responder will be in a consistent state.
Implementation Details  To provide the key-value pair functionality of an authenticated map, we use an instance of an authenticated dictionary (which we will call authdict) paired with a standard java.util.HashMap (which we will call map). To avoid confusion, we will refer to an authentic response returned by authdict as ADAUTHENTICRESPONSE. To insert a key-value pair \((k, v)\) into the authenticated map, we do the following:

1. insert \((k, v)\) into map
2. insert \(k\) into authdict
3. insert \(h(k||v)\) into authdict where \(h\) is any collision resistant hash function and \(||\) is the concatenation operator

If the user queries the authenticated map for a key that is contained, the AUTHENTICRESPONSE returned will contain the key \(k\), the value \(v\) and the ADAUTHENTICRESPONSE generated by performing a membership query on the authdict with the value \(h(k||v)\). The response is checked by verifying the authenticity of the ADAUTHENTICRESPONSE and confirming that the subject of the ADAUTHENTICRESPONSE is the value \(h(k||v)\).

In the event that the key is not contained, the AUTHENTICRESPONSE will contain the key \(k\) and the ADAUTHENTICRESPONSE generated by performing a membership query on \(k\). Recall that since \(k\) was never inserted into authdict, the ADAUTHENTICRESPONSE will verify the presence of two consecutive element \(k'\) and \(k''\) such that \(k' < k < k''\).

STMS implements authdict by means of an authenticated skip list [13], which uses linear space and has logarithmic update time and query time. An alternative implementation of authdict, based on a one-way RSA accumulator, is also available [12]. However, this alternative implementation has inferior performance in practice.

3.2 STMS Systems and Subsystems

As can be seen in Figure 3 the design of the Source and Responder are very similar. At the core of each is the SystemManager which is responsible for loading, starting, and stopping subsystems. The SystemManager is also responsible for instantiating the authenticated map and placing it in a thread safe wrapper. Since the authenticated map is presumably maintaining time-sensitive data, its synchronization wrapper has been written to favor write access over read access.

Every STMS subsystem (on both the side of the Source and Responder) implements the Subsystem interface. This interface provides the common API through which the SystemManager initializes, starts, and stops each subsystem. Each subsystem also maintains a reference to the synchronized authenticated map. It is important to note that the Subsystem interface does not assume any specialization of service; it has been engineered to allow for the implementation of any STMS subsystem.

The SystemManager and the Subsystem API allow for the Source and Responders to be highly configurable and customizable. Every subsystem is packaged in its own Java archive (jar) file which allows for easy system maintenance. It also simplifies the task of adding new subsystems to an already deployed system. To provide for this level of administrative control, the SystemManager can be accessed through the remote interface it provides through Java RMI.

The configuration of the Source, Responder, and their associated subsystems is done through the use of java.util.Properties files. An example Source configuration file in shown in the Appendix (Figure 9). The first two fields define the location of subsystem jars and their configuration files. AuthMapClass gives the fully qualified class name of the authenticated map to use and AuthMapConfFile gives the location of the configuration file used to initialize the data structure. In this case, the source will be deployed using the skip-list implementation described in Section 3.1. For each subsystem which is to be loading, the following is defined:
SubsystemName: A descriptive name which is used for logging purposes
SubsystemCritical: Either true or false: if the subsystem is critical, then its failure to load will prevent the entire systems from loading
SubsystemConfFile: The name of the subsystem’s configuration file
SubsystemLibrary: The name of the jar archive containing the subsystem’s class files

The highly configurable nature of STMS is significant in that it allows for new data structures and subsystems to be developed and deployed without requiring modification to the supporting system code.

3.3 Modification and Query Subsystems
We have developed two subsystems which allow for publication into the source. The first is the simple RMIModificationSubsystem which allows for Java based publishers to access the get and put methods through RMI. We assume that access to this remote object is controlled through external mechanisms, e.g. a firewall which allows access from only the trusted local area network where the source resides.

Also implemented is the LDAPModificationSubsystem which allows the Source to be viewed as a standard LDAP directory. The motivation of this subsystem was to allow Certification Authorities (CA) to seamless publish revocation information into STMS. Further details of CA integration can be found in Section 6.1.

Java clients can query responders through the RMIQuerySubsystem. This subsystem exports all methods relating to queries as a remote interface. In order to provide support for a wider variety of clients, we have also implemented the SOAPQuerySubsystem which allows for Responders to be queried through Web Services. For a detailed description of the Web Services interface to the responder, see Section 6.2.

3.4 Distribution Subsystems
The Source’s distribution subsystem defines a time quantum to determine when synchronization information is distributed to Responders. The time quantum is constant throughout the lifetime of the Source and is defined in the configuration of the distribution subsystem. While authorized publishers can manipulate the data maintained by the Source at any time during a quantum, the changes will not become effective until that quantum expires and the Responders have been updated.

In addition to sending update information, the distribution subsystem is also responsible for creating and distributing a signed basis at the end of every quantum. Before signing, the Basis instance retrieved from the SourceAuthenticatedMap is wrapped in a RichBasis. The RichBasis couples the basis with additional information such as the name of the digital signature algorithm used, the type of authenticated map, and time stamps defining the time the basis was signed and when it will expire.

Responders can join and leave the STMS network at any point during the lifetime of the Source. Since the Source and Responders are synchronized only at quantum boundaries, it is important the Source initialize the new Responder with the state of the authenticated map at the point of the last update. To accomplish this, the method getInitializationData (as described in Section 3.1), is called at the end of every quantum. The initialization data which is returned is cached by the Source and is sent to Responders which join the STMS network during the next quantum.

We have implemented two different distribution subsystems which can be used in STMS. The first is based on Java RMI. The RMIReceiverSubsystem provides a remote interface which the Source can use to send initialization and update information. The corresponding RMIDistributorSubsystem maintains a collection of all active Responders. At the end of each time quantum, the remote
method of each RMIDistributorSubsystem is invoked in a separate thread. This distribution mechanism was designed to support a small number of Responders.

For STMS deployments which expect a large number of responders, we have also implemented a multicast based distribution subsystem. For this we use the Spread toolkit [1] which provides fault tolerant support for group communications. In this subsystem, all responders form a single group to which the SpreadDistributorSubsystem can multicast update information. Each responder also belongs to a private group which is used by the Source to send initialization information.

One important condition should be met for any distribution subsystem: all communication should be pushed from the source to the responder. It is important that the source accept no incoming connections so that it is protected from denial of service attacks. Therefore the source will have to be notified through alternative methods when a new responder wishes to be initialized. Typically, this will involve human intervention.

For testing purposes and ease of deployment, the distribution subsystems we have developed do not strictly enforce this rule. While they allow for notification messages to be received when responders are deployed and taken down, it is a trivial task to remove this feature so that the source is truly isolated from incoming connections.

3.5 Persistence and Crash Recovery

At the end of each quantum, the updates which will be distributed to Responders are serialized to disk and the set is assigned a unique sequence number. In the event that the Source crashes, these serialized update packages can be used to reconstruct the Source from scratch. If the Source does indeed crash, it is highly unlikely to occur in the small window of time after an update package has been committed to disk and before any new modifications are made to the data structure. Therefore, any new changes which occur in the unfinished quantum are lost.

It is also important to observe that the Source crashing does not require the Responders to be restarted. The Responders themselves are not aware of the schedule the Source maintains and are therefore oblivious if a scheduled quantum is missed. Clearly thought, this event will be obvious to any client since the basis will become stale.

The same kind of check-pointing is also done at the Responders after the updates are received from the Source. Since Responders are intended to be inexpensive to deploy, untrusted, and numerous, their periodic failure is expected and does not effect the availability and timeliness of authenticated information. So long as a single Responder remains alive, quantums will continue to pass. Using either of the distribution subsystems described in Section 3.4, the Source is notified when a Responder becomes unavailable and maintains a log of the last update package that Responder received. This is true if the unavailability is caused by either network partitions or by the Responder itself crashing.

After a Responder is introduced back into the STMS, it will most likely be out of phase with the rest of the system (unless, of course, its crash and recovery occur within a single quantum). After reading in and processing cached update packages, the newly resurrected Responder will continue to receive fresh updates from the Source, but it will not commit them into its local data structure until after it has received the update packages it has missed. Retransmission of lost update packages can be automatically handled by the Source during periods of light load.

3.6 Security

The implementation of STMS fully satisfies the following security properties of an authenticated dictionary. These properties rely on standard cryptographic assumptions about one-way collision-resistant hash functions and digital signatures.
Resistance against compromised responders. A compromised STMS responder cannot convince a client to accept an incorrect answer to a query.

Resistance against active adversaries on the network. An adversary that intercepts and modifies an update received by a responder or an answer received by client cannot cause a client to accept an incorrect answer to a query.

In the above scenarios, the client will recognize that the answer is incorrect because the basis computed from the proof and answer will not match the basis signed by the source.

4 Performance

This section describes the results of our experiments on the performance of the core authenticated map and the behavior of the system when deployed across a wide area network. Information on the types and locations of the machines we used is shown in Figure 4. All the experiments described below used 160-bit keys and values. The data structure used was the authenticated map discussed in Section 3.1.

<table>
<thead>
<tr>
<th>name</th>
<th>location</th>
<th>os</th>
<th>processor</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mach1</td>
<td>Brown University</td>
<td>Windows XP</td>
<td>dual 2.2 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>mach2</td>
<td>Brown University</td>
<td>Windows XP</td>
<td>dual 1.7 GHz</td>
<td>1 GB</td>
</tr>
<tr>
<td>mach3</td>
<td>Stanford University</td>
<td>Linux</td>
<td>1.8 GHz</td>
<td>1 GB</td>
</tr>
<tr>
<td>mach4</td>
<td>UC Irvine</td>
<td>Linux</td>
<td>1.8 GHz</td>
<td>512 MB</td>
</tr>
<tr>
<td>mach5</td>
<td>UC Irvine</td>
<td>Linux</td>
<td>1.8 GHz</td>
<td>512 MB</td>
</tr>
<tr>
<td>mach6</td>
<td>UC Irvine</td>
<td>Linux</td>
<td>1.8 GHz</td>
<td>512 MB</td>
</tr>
<tr>
<td>cluster</td>
<td>Brown University</td>
<td>Linux</td>
<td>1.8 GHz</td>
<td>512 MB</td>
</tr>
</tbody>
</table>

Figure 4: Machines used in our experiments, where “cluster” refers to a collection of 21 machines.

4.1 Data Structure Performance and Query Throughput

Performance results for the authenticated map were conducted on mach1. As Figure 5(a) shows, all of the operations we tested took on average less that one millisecond. Of particular interest are the results for queries and validation which consistently remained under 200 microseconds. The query results measured the time to produce an AuthenticResponse (as would be done by STMS Responders). The validation results demonstrate the amount of work required by a client. These times do not include the time to verify the digital signature on the basis. We do not add the cost of a digital signature to the validation time since our architecture allows for many queries to be validated against a single basis.

We have also done experiments on how the data structure performs when deployed within a STMS Responder. A responder, running on mach1, was initialized with one million key-value pairs. Client processes running on each of the machines in the cluster at Brown University queried the responder through its RMIQuerySubsystem. Each client process queried the Responder with random 20-byte arrays as quickly as possible, performing either one, five, 10, or 15 queries for each remote method invocation.

Figure 5(b) shows how performing multiple queries for each connection greatly minimizes the overhead of the remote method invocation. When 15 queries per connection are made by the client applications, the Responder throughput nearly matches the performance of the raw data structure.
4.2 Distribution Subsystems Performance

Figure 6 shows the performance of our two distribution subsystems when used to initialize Responders. They show the throughput that each protocol achieved on both the local area network at Brown University and across the Internet. Since this experiment was used to test initialization, it was sufficient to use only a single responder. For both the local and wide area network tests, mach1 was used as the STMS Source. mach2 and mach3 were used for Responders on the local and wide area test, respectively. RMI was clearly superior to Spread in terms of unicast communication in both the local and wide area settings. This was to expected since RMI does not incur the significant overhead of providing multicast services.

On the other hand, the benefit of using a multicast toolkit for updating Responders can clearly be seen in Figure 7. In this experiment, 21 machines in a cluster at Brown University were used. One machine ran the source and updates were distributed to between 1 and 20 responders. The size of each update varied between 10, 30, and 50 thousand elements (7.1, 21.2, and 35.3 MB respectively).
For each of the three different update sizes, Spread’s transmission time remained consistent irrespective of the number of connected responders. But, as the figure clearly indicates, the RMI distribution subsystem shows update times which are proportional to the number of connected Responders.

These trends have also been verified when STMS is deployed across the Internet. In our wide area network tests, mach1 at Brown University was used as the Source. The machines at Stanford University and UC Irvine were used as Responders. As in the local area network tests, RMI outperforms Spread only when a small number of Responders are connected to the Source. As Figure 8 further indicates, the RMI distribution subsystem update times grow linearly with the number of connected Responders while Spread’s communication overhead is largely constant.

4.3 Time Quantum and Quality of Service

Since STMS relies on a time quantum to schedule updates to the untrusted responder, it is important to understand how the length of the time quantum will effect the ability of the responders to
provide up-to-date information. We observe that for any reasonable deployment, the time quantum must be significantly longer than the time to propagate and apply the updates.

At the start of a new quantum, the basis is time-stamped and digitally signed by the Source. The "valid from" field of the basis is set to \( t \) and the "valid until" field is set to \( t + q \) where \( t \) represents the current time and \( q \) is the length of the quantum. During the time it takes to distribute the newly signed basis and the changes to the dictionary, Responders will be providing stale information since its data structure reflects the state of the previous time quantum. We leave it up to a client’s locally defined security policy to decide if the stale response is sufficient or if it should query again at a later time. While the new updates and fresh basis are being committed, the responder will be unable to answer queries since the state of the data structure will be inconsistent with the basis.

The most appropriate deployment environment for STMS is one in which updates are infrequent, the quantum length is long, and the query rate is very high. In this case, the update propagation and commit time will consume a small percentage of the time quantum. This will allow responders to provide fresh responses the majority of the time. Also, since the quantum is long, clients will be able to make many queries while only having to retrieve one basis (and therefore needing to verify only one digital signature).

5 jssh: a Secure Shell Client Based on STMS

Our architecture for trusted data distribution has many applications. In this section, we describe the implementation of jssh, an SSH version 2.0 client that uses STMS for authenticating the public key of an SSH server.

The SSH protocol [22, 20] allows two hosts to communicate securely over an untrusted network. Once established, an SSH connection provides both user authentication and data transfers with encryption and integrity protection. These protection mechanisms are used to implement remote terminal sessions, network port forwarding, and file transfers.

The protocol employs a client-server communication model. Upon establishing a connection, the client and server exchange protocol version information and negotiate to determine which key exchange, public key, encryption, integrity, and compression algorithms will be used. In each of these categories, lists of supported algorithms, ranked in order of preference, are sent to the other side. If possible methods amenable to both sides are selected. Otherwise, the session is terminated. The SSH Specification requires that all conforming SSH implementations provide a set of standard algorithms to ensure interoperability.

The initialization vectors and session keys necessary for the encryption, integrity, and compression algorithms are derived from the key exchange method. The required key exchange method is known as the diffie-hellman-group1-sha1 algorithm. This key exchange method places the integrity of the key exchange and hence the security of the connection on the client’s ability to verify the authenticity of the SSH server’s public key. The SSH Transport Specification provides a detailed description of the computations and messages used by this key exchange method.

To provide the critical mapping from servers to public keys, SSH clients may use either a local or global database. By local, we mean a database created and maintained by an individual SSH client. In contrast, a global database is maintained by a trusted third party and is available for consultation by multiple SSH clients.

Of these options, a local database is the simplest to implement since it does not require a system to distribute the key information. As a result, local databases are by far the most common method employed by SSH clients to store this information. Unfortunately this approach exposes the client to attack by malicious adversaries.
When using a local database, a client’s initial connection to an SSH server is susceptible to attack. The first time a client connects to a server the host name to public key mapping is unknown. With a local database, the only option is to accept and store the association on the first connection. If an active adversary performs a man-in-the-middle attack during the initial exchange, the adversary can subvert subsequent security mechanisms. Since the mapping is stored in the local database and is therefore persistent across connections, the adversary will be able to compromise future sessions as well.

Even if the initial connection is not compromised, there is a drawback to storing the information in a persistent local database. If the server’s public key should change at some future date, there is no convenient way to inform a client. The only mechanism provided within the SSH protocol is to send the server’s new public key to a client during key exchange. From the client’s perspective, there is no way to distinguish this event from an active adversary’s attempt to compromise the connection. The client must depend of the user’s judgment to decide whether or not to accept the new key. Since many users are not familiar with the ramifications of accepting a new key, this solution presents another security problem.

In addition to the conceptual drawbacks of the local database model, the typical implementation also has deficiencies. The standard approach is to use a file to store the database and use file permissions to prevent access or modification by other users. However, a file is not always the correct abstraction for storing such a database. If the operating system does not provide file permissions then any user on the system will be able to corrupt the database. Since not all versions of the popular Microsoft Windows family of operating systems support file permissions, this problem is significant. If the SSH client allows users to store the file on a network file system, then the file is exposed to network attacks. Since most large multi-user systems use network file systems to store home directories without any network layer security, this is a common deficiency. Finally since the file’s format is not part of the SSH standard, no two SSH clients are expected to store the data in the same manner. This presents a deterrent to switching clients. All of these problems demonstrate that files provide a poor choice for storing the database.

Our SSH client, jssh, uses STMS to store the mapping from an SSH server’s host name to the server’s public key. Using STMS, the exchange can be made secure without any modifications to the existing SSH protocol.

For each SSH server with which users will communicate, the necessary information must be stored in an STMS map. For each server, two pieces of information are inserted into an authenticated map: a mapping from the SSH server’s host name to its public key and another mapping from the SSH server’s IP address to its public key. By storing mappings for both the host name and IP address, users are given the option of typing either to specify the server. A separate utility was created to perform this step.

Before establishing an SSH connection, jssh allows users to select the method for authenticating the SSH server’s public key. If the user wishes to consult an STMS map, the user is able to specify the name of the STMS responder to consult. When jssh receives a server’s public key, the client queries an STMS responder using the RMIQuerySubsystem. The response is then verified and the mapping validated.

This design has several advantages over using a local database. The procedure allows for a completely secure key exchange, eliminating any opportunity for attack. Also, the procedure allows keys to change dynamically from session to session. Since no file needs to be created, the associated security risks are not present. This allows for clients to be implemented correctly on a number of common platforms. Since the database is global, administrative task are centralized. For all of the reasons given above, this solution is therefore much more scalable and secure than the local database approach.
6 Other Client Applications of STMS

In this section, we overview several other applications that we have implemented using STMS. With each of these applications, the existence of STMS allowed to produce working clients very quickly. To evaluate each application, we describe the sources, the users and the application of the data distributed with STMS.

6.1 Certificate Revocation

Our work was originally motivated by the certificate revocation problem; hence, it is natural that our first application is for certificate status checking. A STMS-based certificate revocation system works as follows: the CA publishes certificate revocation information to STMS as certificates are revoked. Certificates are identified by concatenating the name of the CA and the certificate serial number then digitally hashing this information. A certificate revocation query then becomes a query to a STMS responder, checking for the existence of the hash of a CA, serial number pair. As a specific client application, we have integrated STMS with the RSA Keon CA (see www.rsa.com).

RSA Keon uses an LDAP server to publish certificates and certificate revocation information. Instead of having the certifying authority (CA) connect to a standard LDAP server, however, we configured it to connect to an LDAP-based publication subsystem integrated into STMS. Thus, whenever the CA connects to the LDAP server to publish a certificate revocation list, the STMS LDAP subsystem iterates through the list and updates the repository accordingly.

In order for this picture to be complete, we must address the issue of querying for certificate status. Previous versions of STMS have supported a modified version of the Online Certificate Status Protocol (OCSP) on the responder side. This subsystem could receive a standard ASN.1 encoded OCSP query and would return a non-standard algorithm-specific OCSP Response. Given the cumbersome nature of ASN.1, maintenance of the OCSP query subsystem has been discontinued, however. The current version is implemented by integrating our certificate status checking into another client application we developed, which is based on the XML signature specification. We describe this application next.

6.2 STMS and Web Services

The term “Web Services” is generally used to describe a collection of protocols and standards that are used to facilitate interoperability between clients and servers. One of the major factors for their success is the fact that they are built upon existing Internet standards such as XML and HTTP. SOAP, the Simple Object Access Protocol, is an XML-based protocol for packaging messages and facilitating RPC-style communication between clients and servers. By building a Web Services interface to the STMS Responder, we are able to better utilize existing standards to achieve greater levels of portability, interoperability, and ease of use for client applications.

We have developed full client toolkits which allow for the rapid development of Web Service based STMS clients. These toolkits allow clients to communicate with STMS Responders using SOAP and facilitate the verification of STMS proofs using either Java or C#, the native language for Microsoft’s .NET framework.

Our web services subsystem also allows for STMS proof verification to be accomplished through the use of standard XML Signatures and a custom XML transform. The XML transform is responsible for verifying that the proof returned by the Responder is consistent with the basis. This transform is automatically applied through the standard technique of verifying an XML Signature. As a result, the computation of the proof and the verification of the basis’ signature is performed in a single step from the perspective of the client.
Further details regarding STMS's use of Web Services and XML Signatures can be found in [19].

6.3 End-to-End Document Integrity

Another application we have developed is to use STMS to validate the integrity of documents downloaded over the web. STMS is used in this case to validate that the document received by the client is identical to the document loaded into the web server. Current commercial integrity products verify that a document leaving the web server is the same as the document put in. Therefore, these systems do not detect tampering that occurs during transmission. A common example of an attack not detected is DNS redirection where a URL is hijacked to a different server. The attack is not detected because the page is served by a different web server. The STMS-based integrity system will detect these kinds of attacks.

The STMS-based integrity application is associated with a standard HTTP web server, serving a collection of documents. STMS can monitor entire documents, or in the case of dynamic content, only the changing portions of documents. STMS contains two entries for each document (or sub-document): one a cryptographic hash of the document URL and the other a cryptographic hash of the URL combined with the content of the document. We use a change detection agent that monitors the HTTP server’s document collection. When a change (a document creation, deletion, or modification) is detected, the CDA immediately updates the STMS repository.

On the client side, a standard web browser serves as the user’s primary interface to the end-to-end integrity system. We have developed a tool bar integrated into the Internet Explorer browser (implemented using the .NET framework). The STMS integrity toolbar responds to browser events by making queries to the STMS responder. Users browse pages as they normally would. Each time a document is loaded, the toolbar automatically performs an integrity check.

The integrity check consists of two steps. First, the toolbar determines if STMS has an entry for a given page. This is done by computing the hash of the URL alone and querying the responder for this value. If no entry is found, then the document is not under STMS control and no further action can be taken.

If the URL is found in the responder, then the toolbar locally calculates the hash of the document (or sub-document) and queries the responder with this value. Three results are possible. If the document fingerprint is contained, then the integrity has been verified successfully. Conversely, the absence of this element indicates the integrity of this document has been violated. The third possibility is that, based on the last update time of the responder, it is possible that integrity information has not yet propagated to the responder. The result in this case is a temporary “status pending” state and the user is notified that the integrity check cannot be performed at this time.

There are many applications where this type of query. For example, a financial speculator that receives NASDAQ stock quotes from the Yahoo! Finance Web site would be well advised to obtain a proof of the authenticity of the data before making a large trade based on that information. Another application would be to guarantee the end-to-end integrity of dynamically generated web content from geographically distributed mirror sites.

6.4 Personal Digital Media Library

In recent years, high-speed Internet connections have become increasingly available across the country, and media formats such as MP3 and MPEG have enjoyed an explosive growth. Unfortunately, much of this trade involves the pirating of copyrighted music and video, prompting loud protests from the music and film industries. These groups insist that this burgeoning market for digital media exchange be legitimized.

The music and film industries are calling for is a system where a vendor can sell non-duplicatable rights for music or video to a user. Such a system cannot be successful unless it meets the following
criteria:

- Users must be able to conveniently purchase right to media.
- Access to purchased media must be as convenient as possible.

We have built an application called the Personal Digital Media Library (PDML) that demonstrates how STMS can be used to help solve these problems. PDML distributes digital files from a file server to the user. The purpose of the PDML client is to validate the user and the file. In particular, STMS is involved in three important security processes: validating the user, validating the user’s rights to a particular media file, and verify the integrity of the file.

STMS is used to verify or validate three kinds of data:

- **Identity certificates** - We have implemented a lightweight authentication system based on digital certificates identifying each user. STMS is used to check the revocation status of these certificates.

- **Attribute certificates** - We use attribute certificates to indicate a user’s rights to a particular media file. These lightweight certificates are not digitally signed. The certificate is validated by its inclusion in the responder. This architecture greatly reduces the administrative overhead associated with ordinary digital certificates and so scales up to a very large number of certificates.

- As with web content, STMS can be used to check the integrity of the media files. This is especially important when files are server from a peer-to-peer network where files can become corrupted as they move from user to user.

Sample screen shots of the PDML application are shown in the Appendix (Figures 12–14).

**Acknowledgments**

We would like thank several people for contributions to the development of STMS and its client applications, including Andrea Carmignani, David Emory, Benety Goh, Dan Polivy, Jeremy Mullendore Joel Sandin, and Andy Schwerin. We would also like to thank Steven Barron, John Garrett, Richard Sneider, Thorne Sparkman, and William Winsborough for several useful discussions regarding the topics of this paper.

**References**


A Appendix

\begin{verbatim}
stms.source.SubsystemLibDir=../lib/source
stms.source.SubsystemConfDir=../etc/source

stms.source.AuthMapClass=stms.authmap.asl.ASLSourceAuthenticatedMap
stms.source.AuthMapConfDir=.
stms.source.AuthMapConfFile=aslauthmap.properties

stms.source.SubsystemName.0=RMI Distributor
stms.source.SubsystemCritical.0=true
stms.source.SubsystemConfFile.0=rmidistributor.properties
stms.source.SubsystemLibrary.0=rmidistributor.jar

stms.source.SubsystemName.1=RMI Modifier
stms.source.SubsystemCritical.1=true
stms.source.SubsystemConfFile.1=rmimodifier.properties
stms.source.SubsystemLibrary.1=rmimodifier.jar
\end{verbatim}

Figure 9: An example configuration file used to deploy the Source
Figure 10: Connection screen of jssh.
Figure 11: Transport screen of jssh.
Figure 12: If the Identity Certificate stored on the user’s smart card has been revoked, the user is denied access to their media.

Figure 13: Songs that a user has purchased rights to appear in the user’s media library.
Figure 14: Every time a piece of media is requested, the status of the media’s corresponding attribute certificate is checked by querying an STMS Responder. Since media is possibly purchased from many independent vendors, this level of fine-grain access control is essential.