Media Bank: Access and Access Control

by

Derek Allan Atkins

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
May 1993

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author ....
Program in Media Arts and Sciences
22 May 1995

Certified by ....
Andrew Lippman
Associate Director, Media Laboratory
Thesis Supervisor

Accepted by ....
Stephen A. Benton
Chairman
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Media Bank: Access and Access Control

by

Derek Allan Atkins

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on 22 May 1995, in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES

Abstract

The media bank project at the Media Lab is a distributed multimedia object database which allows clients to produce and consume synchronous data over a data network such as the Internet. Each server in the media bank can deliver data objects to clients who request them. This thesis proposes to solve a problem arrising from this system's openness: any client can obtain any object from the media bank at any time, and anyone watching the network has the ability to not only read the data but insert anything an attacker could want. Moreover, running the media bank software must not open any security holes in the servers. In a commercial environment, the original design of the media bank just would not do. A set of security services for the media bank would solve these problems.

The media bank security services start with a secure server which can provide a secure network connection to a client. By encrypting the data, network attacks can be drastically reduced. Moreover, a digital movie ticket provides access control to the media bank data. A digital movie ticket is a voucher token that can only be used for a specific transaction. A client obtains a voucher for a specific set of data by purchasing that voucher from the media bank. In this manner, a client is limited by the voucher to retrieve only the data for which she has paid.

Thesis Supervisor: Andrew Lippman
Title: Associate Director, Media Laboratory

Media Bank: Access and Access Control

by

Derek Allan Atkins

The following people served as readers for this thesis:

Thesis Reader . . . .

Kenneth B. Haase
Assistant Professor of Media Arts and Sciences

Thesis Reader . . . .

Daniel Schutzer, Vice President
Director of Advanced Technology
Citicorp

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed software systems require distributed security solutions. In a distributed environment clients and servers speak to each other over a shared resource, the network. Because many people use the same network, servers are required to have some trust model which defines how to authenticate and authorize clients. Many authorization models have been tried and tested in the past, and new models are being tried and tested today. This thesis describes a model of security and authorization for a particular application, the media bank project at the MIT Media Lab.

1.1 Background and Motivation

The Media Bank Project at the MIT Media Lab [11] is an attempt to add synchronous data types to an environment like the World-Wide-Web. The Web is very good at providing static data, such as text and images, and provides hypertext links between different data to allow a document to be presented in a non-linear fashion. However the Web fails when data needs to be synchronized together, such as audio and video segments in different datafiles, or, worse, a program that is created by combining many datafiles, such as multiple streams of video, audio, and other information. The media bank proposes to solve this problem (see chapter 2).

The basic building block of the media bank is an object. An object is data that contains some internal coherency. For example, data in one object could describe data in another object. The media bank is really just a collection of objects and a set of methods to access those objects.

The building blocks of the objects that comprise the media bank are *dtypes* [4]. Dtypes are host-independent, network-independent objects that can be created, accessed, transmitted, and destroyed through multiple interfaces. In particular, dtypes can be accessed through C++ or Scheme, a LISP dialect. Also, methods exist to transmit dtypes between machines via the network.

The most useful thing about the dtype system is that it contains an embedded evaluator. Using the dtype evaluator, a programmer can define a set of primitive functions and allow a Scheme script to control the execution of the program. However, this leaves a dtype server open to attack, and a machine running such a server insecure.

In the normal operation of a dtype server, a Scheme evaluator is instantiated and then the server awaits client connections. When a client connects to the server, it has access to that evaluator with a shadow of the server's environment. Any commands the client wants to execute will dutifully be performed by the server; there is no restriction on the commands a client can run or what data structures a client can access. For example, commands such as reading and writing any arbitrary file (like the server's password file) are available to any client connecting to the server. Worse, a client could rewrite and replace one of the server's functions so later clients would see the wrong procedure.

Another problem with the dtype system is that all data is transmitted in the clear over the network. Anyone listening to the network can read all the data traveling over it. An intruder could *sniff* the network, pick up interesting pieces of data, and use that data to his advantage. There is no confidentiality of data traveling the wire; anyone can read it. While all these problems are not unique to the dtype system or the media bank in general, running a dtype server opens up more security holes that can be used to compromise a machine.

Because the media bank is comprised of many objects, as opposed to a single stream for a single program, it provides a different paradigm of on-demand services. As a result, new ways of looking at security need to be devised.

In general, security systems consider specific use patterns. One pattern is where a single security token is good for a single purpose. Another pattern has general tokens where a single token is good for everything. Neither approach works in the media bank. This thesis examines the possibility of securing the media bank with a single token, multiple purpose security system.

1.2 Theoretical Example

Assume that media bank technology is widely used in the future. The neighborhood Blockbuster uses the media bank to supply home video to its customers. Museums use the technology to catalog and store art and literature. Advertising firms use the bank to create references to other material for sale. And through it all, clients pay for the services that the media bank renders.

If some user wants to obtain some file on Blockbuster's server, all he needs to do is connect to the server and execute a command to read that file. The server would dutifully interpret the Scheme code and open the file, allowing the client to read anything that Blockbuster has on their server. The client only needs to know what files exist. Yet since the dtype server allows execution of Unix commands, a user need only ask for a file listing and that information would be found. Limiting the client's ability to execute arbitrary functions would solve this problem.

Another problem is if some user in one house wants to watch some movie, there is nothing to stop someone in another house down the road from watching it, too. Since the data is transmitted in the clear over the network, anyone can sniff the data as it flies by. Programs can be written to store, process, and use the sniffed data as well. What if the data being transmitted were financial in nature, a digital cash token or a credit card number? The network sniffer can pick up the data and an attacker could then use that information. The worst part of this problem is that there is no way to know if someone else is eavesdropping on a conversation.

Another related problem is a more active attack, spoofing network packets. A user could spoof a packet to Blockbuster, making it appear to come from some advertiser, such that Blockbuster would perform a service for the advertising firm. Then the attacker can reap the rewards of the service, watching the resultant data, and neither Blockbuster nor the advertiser would know who did it.

An attacker could theoretically take over an existing network connection. If some researcher were connected to a museum server, an attacker could use a program to rewrite the researcher's commands to the server, completely invalidating the research. Worse, the attacker could get the museum server to execute functions, pretending to be the researcher, and watch the results. The researcher would be blamed and charged for the usage.

What if some family wants to watch a movie from Blockbuster, pays for it, and is given some code to use to actually view the movie? There is no mechanism to stop the family from watching the movie twice, or three times, or watching it every day for a week. Someone sniffing the net could see the token and then they could watch the movie as well. Some mechanism needs to be in place to stop all these attacks.

1.3 Security Services

In order to protect servers and data, a set of security services were developed to augment the dtype system. Primary motivations were to limit the access a client could achieve. Moreover, protection from network sniffing attacks was a must.

1.3.1 Securing the Server

One set of security services, the *secure evaluator* (see chapter 3) was developed to protect the server machine and the data traveling over the network. This service limits the client access to the server by restricting the client environment. A client can only run the functions a server allows it to run. In this manner, a server can be sure that a client cannot use the Scheme evaluator to attack the machine.

Yet attackers can use other means to break security, such as network sniffing or actively spoofing network packets, rewriting data to make it appear to be something else. Just by sitting on the wire and watching network transmissions, an attacker can usually gain enough information to illicitly obtain services. For example, an attacker could use a network sniffer to watch a user type a password, he could watch a filehandle cross the net, he could even employ a program to copy the data from an object as it flies across the wire. Worse, an attacker can take over an existing network connection by running software to spoof network packets, to make it appear as if the client changed locations, or grab some data mid-stream and change its contents.

All of these attacks need to be deterred, and the secure evaluator provides a mechanism to do so. In particular, it can protect all the data crossing the wire by using various forms of cryptography. Data can be protected from spoofing or change by providing a data integrity mechanism. One such mechanism is a digital signature, where a message is signed using a cryptographic protocol so that anyone can verify the origin of the message. If a client and a server both digitally sign messages, then both can be assured that data has come from the other without tampering.

Data can be protected from spying eyes using another form of cryptography, encryption. Only the entities with the proper keys can decrypt and read an encrypted message. Without the appropriate keys, an encrypted message appears to be completely random bits.

To realize the secure transmission of media bank data, a transport server was created (see chapter 4). This program allows clients to connect and download data from the media bank in a secure fashion. It uses the secure evaluator, which restricts client's access to various commands, to protect the server from possible attack. However, it accepts a limited set of commands from clients to access objects stored in the media bank.

Whatever data protection the client chooses, the identity of the user is hidden; there is no need to know who is talking. Both entities, client and server, are only guaranteed a continuity of conversation. In other words, the same entity is sending requests, the same server is responding to them. At no time does a server need to know who some client is, or that she really is some specific individual.

1.3.2 Securing the Data

Another service, the *digital movie ticket* (see chapter 5), was created to protect the data that is served using the transport servers. While encrypting the data sent across the net is a good idea in general, it doesn't solve the problem of a legitimate client downloading data for which he is not authorized to obtain. Perhaps the user is only supposed to download the data once. The digital movie ticket is designed to solve this problem.

The idea is that a user can obtain a ticket from a dedicated server and use it to authorize the download of all the objects that comprise a media bank program. The ticket is used as an authorization token allowing the client to download a particular object or set of objects. The token can be checked by the transport server to see if it has already been used. If it has been used the access can be denied, insuring that the client downloads the data just once.

The ticket itself can be purchased using some form of digital cash. Alternatively, objects can be flagged as `free` objects, which means that they do not cost anything to obtain. The transport server can decide if it requires payment for its data.

Since digital cash protocols are anonymous, the movie ticket server has no concept of identity. Again, the media bank servers do not need to know who a client is in order to serve him. All they need to see is a valid movie ticket, and the servers know that the client has been authorized to download that object.

However the movie ticket cannot be used just by itself to protect data. Mere posession of a valid movie ticket provides enough authorization to download data. Therefore, the movie ticket must be protected while travelling the network. This is accomplished by encrypting the link using the secure evaluator in the server. Using the secure evaluator also prevents replay attacks, so the movie ticket server is safe from an attack mimicing a ticket request.

Chapter 2

The Media Bank

The media bank is a distributed set of servers that provide multimedia data objects and information about them. The system is implemented as a number of object servers and a central database server, which keeps track of all the data in the system. A client requests information about an object from the database server and requests more information from the object servers. This chapter describes selections of the design and implementation of the media bank in greater detail.

2.1 Design

The media bank is designed as a distributed client-server system. The system is a hierarchy of servers consisting of a directory server and a number of item servers (also called an object server[1]). The item server runs on every machine that serves data in the media bank. The directory server connects to each item server and maintains a

[1] The original name of the server was `item server`, however the name is being changed to `object server` because of expanded functionality. The terms are used interchangeably throughout this document.

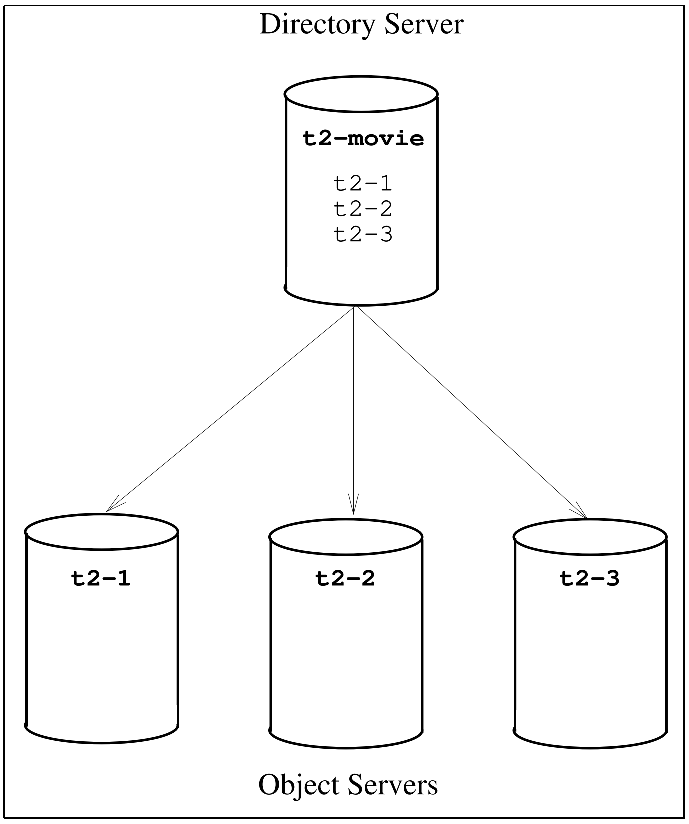

Figure 2-1: The directory server contacts the object servers to locate all media bank objects.

list of all objects in the media bank. More information about these servers is located in sections 2.3.1 and 2.3.2.

A client contacts a directory server, which contains a list of all the objects in the media bank, to find out information about a particular object. This information includes data such as the location, which consists of a hostname and a port. The client then uses that information to contact the item server to get more information about the object, or to retrieve the data it contains (see figure 2-1).

The servers and clients should not depend upon any specific hardware. One of the goals of the media bank is to provide a system where intelligent clients can choose the data they require, and an unintelligent client can request the server to perform its manipulations. An intelligent client is one that knows something about itself, such as what hardware is available or what formats are preferred over others. Data can be stored in multiple formats, so a client which knows the hardware on the system can choose the data format which best suits its needs.

Another goals of the media bank is to provide replication of data. A single data object can be located on multiple servers. This provides redundancy as well as load sharing, so a client that fails to retrieve an object from one server can jump to another server to obtain it.

2.2 Dtypes

Dtypes are a platform independent means to define objects and pass them around between processes and machines on a network. The core of the dtype system is a set of primitive objects which are used as building blocks to create more complex data structures. These concepts also allow multiple-language representations of objects, and provide for easy communication between programs independent of the hardware platform. The current dtype system provides for C++ and Scheme definitions of objects. A list of some of the dtype primitives is in table 2.1.

A complex structure can easily be constructed by combining primitives in certain manners. For example, a typed structure could be implemented as a vector of pairs, where the first part of the pair is the name of the element, and the second type is the value. For example, a timestamp object, which adds the current time to a piece of data, could be implemented in this manner:

```
#(
   (name . timestamp)
   (timestamp . current-time)
   (data . object-being-stamped)
)
```

Indeed, it would be simple to add other information to this object, to build upon

null	A Null, used to end objects
boolean	A simple true or false value, denoted as `#t` and `#f`
voidptr	A pointer to an arbitrary piece of C++ memory
symbol	A symbol is a pointer to some Scheme value
string	A series of characters, denoted as `''value''`
pair	Two objects joined together, denoted as `(obj1 . obj2)`
list	A series of objects created by subsequent pairs and ending in a nul. Denoted as `(obj1 obj2 ...)`
vector	A numbered list of objects, denoted as `#(obj1 obj2 ...)`
number	A set of different types of numerical representations, including integers, rationals, and reals, denoted by some representation of the value
compound	A means to create arbitrary dtype objects. This is a pair where the first part is a symbol and the second part is some internal representation of the object

Table 2.1: Many of the dtype object primitives.

it, in order to suit most any need. The dtype system allows objects to be built upon each other to achieve an end.

2.3 Implementation

This section describes the implementation of the Media Bank servers, the directory server and the item server. Both servers are written in Scheme and perform a set of functions. These sections will explain what the servers were originally implemented to do, and how they can change to support new functionality.

2.3.1 Directory Server

The directory server's role is to keep track of all the objects in the media bank. It performs this by contacting all the item servers and downloading their database serial numbers and the names of all their items. The serial number is used to see if the data has changed; when the serial number increases, then the data needs to be reread. Different items with the same name are assumed to contain the same information, therefore equal names are combined into a single object for clients.

When a client makes a request of the directory server, it asks each item server for its current serial number. If any serial numbers have changed the directory server downloads the list of objects from the server that changed. Once all the item servers have been checked, the directory server will honor the client's request. In particular, the directory server will tell the client what objects are available, and where they are.

In the future, the directory server should also include some amount of information about the objects. For example, a client should be able to obtain the format information about an object, and perhaps even the content information from the directory server. This way the client only needs to ask the directory server about an object, rather than recursively tracking the object through all the item servers on which it resides.

2.3.2 Item Server

Once a client determines where an object is located, it contacts the item server on that machine. Currently, the item server contains the following meta-information about an object: name, format, size, encoding, location, etc. This information is useful to a client that is trying to determine which objects to download and how to download them.

The item server needs to know all the objects that it exports to clients. Currently this is done by a flat file that is loaded at run-time. Unfortunately this method is very static. It is difficult to keep the meta-information synchronous with the object data when the two are kept separate. For example, if an object changes location or the data in the object changes, someone has to modify the file of meta-information to keep it consistent. It would be much better if the meta-information actually resided with the object itself.

The next version of the item server will actually implement this. Each object will contain a file named `descriptor.dtype` which contains the meta-information about the object. The new version of the item server will look for all these files and store them in its database. In this manner it can find all the items it serves dynamically, rather than depending on a file to list all the objects. This also allows easy inclusion of new objects, since just placing an object and descriptor in the filesystem will allow an item server to find it.

Currently, a client only talks to the item server to find out how to download an object and where it is located. This too will be changed in the new version. The item server will become an object server. In other words, it will be given the ability not only to distribute the meta-information about an object, but to serve the object data as well. For a discussion of the protocols involved in the object server, see chapter 4.

One problem with the current implementation is that an item server can export an object that actually resides somewhere else in the file system because media bank data is obtained via NFS. A client is told where in NFS to find an object and it looks for it in the filesystem. In one instance, an item server was advertising an item which was sitting on the local disk of another machine. The result was that only clients on the particular machine that had that disk could read the data as advertised. This, too, will change in the new item-server, since it will look for items in its local filesystem. The server can be told where to look for objects and will only find objects that it can actually read.

This thesis involves making the changes to the item server to turn it into an object server. In addition, small changes are required of the directory server to enhance the security of the media bank system as a whole.

Chapter 3

The Secure Evaluator

The dtype package provides a Scheme evaluator which allows clients to initialize and modify state on the server. The server sets up an initial environment, which each client sees on connection. The client can then modify that environment, create objects, define procedures, etc. While this might be acceptable for some servers, it is not acceptable for the media bank. What is needed is a means to allow the client to only access predefined symbols in a secure manner. The result is the secure evaluator.

3.1 Motivation

An evaluator is a procedure that interprets and executes commands. In normal operation, the evaluator listens for input, reads in data, parses it, and then acts on what it finds. Normally the evaluator has some environment in which it operates. When it finds a token, it looks up the value in the current environment. For example, if the evaluator were given

```
(+ 2 3)
```

it would parse the `+` as a function, look it up in the environment, and then call that function with the arguments `2` and `3`. This lookup is called a binding, so the symbol `+` is bound to the function of addition.

The default environment contains a function called `define`, which allows a user to bind data to arbitrary symbols in the environment. In fact, it is possible for a user to redefine a symbol and bind it to some other value. For example, a user could redefine the symbol `+` to perform subtraction. This would cause all future evaluations of this symbol in this environment to perform the new function binding.

While this functionality is useful in concept, it is disastrous to a server trying to protect data. Instead, a means to limit the abilities of the client is needed, to protect the server from rogue or careless users. If the evaluator can limit the functions that are defined in a client's environment, then the server can dictate the functions that users are allowed to execute. This is the basis for the secure evaluator.

3.2 Design

The secure evaluator provides two means to protect a server. First, it limits the symbols in the server environment. This means the client can only execute a limited number of functions. Second, the secure evaluator provides for link encryption, which means that all network traffic between the server and client is encrypted in some agreed-upon method.

The first design consideration for a secure evaluator is a limited environment. The server needs to decide which functions are exported to the client, and which functions only the server process should be allowed to execute. Since the server is most likely written in Scheme, there needs to be a mechanism whereby the environment of the client can be deterministically set by the server, instead of the client just obtaining a shadow of the current server environment, whatever that might be at the time.

Once the server has limited the functions a client can evaluate, all other known attacks are limited to holes in the base operating system, network sniffing, or network spoofing.[1] When the normal dtype network protocol is in use, someone could monitor the network traffic and read the dtype data traveling across the wire. Worse, an attacker could even take over the network connection, spoofing himself as the real client. Since data is sent over the net in the clear, the server has no way to prevent this attack, and the spoofer can gain unauthorized access to data.

To prevent these forms of attacks, the secure evaluator provides a mechanism to encrypt the dtype data traveling across the network. A client can set up a secure connection using a key exchange protocol, which will create a context on the server containing the appropriate security measures. From that point the server will expect all further communications to be encoded using the mechanisms agreed upon in the key exchange, and it will encode responses using the same mechanisms. This scheme protects the data from a network snooper, since anyone watching the bits will get a bunch of gibberish.

In addition to just encrypting the data, it can be signed using digital signature technology (see section 3.3.4). Using a digital signature, one side of the connection knows for sure that the other side was actually the sender. A signature that doesn't verify properly is erroneous and should raise an exception. However, a message that is signed but does not have a verification method should be passed on without checking the signature. For example, the server can sign all its outgoing packets, but the client can choose whether or not to verify the signatures.

[1]A clever hacker can break into the operating system of a server and read or modify data in that manner. Those attacks are not considered in this thesis.

3.3 Dtype Cryptography

The first step in creating a secure Scheme evaluator is adding cryptography to the dtype system. Since the dtype system is an interface between Scheme and C++, the most reasonable design would provide the primitive cryptographic functions, and allow users to put together more complicated systems by combining the primitives. The primitives involved are random number generators, hash functions, block ciphers, and public key ciphers.

The plan is to take a pre-existing cryptographic library and wrap Scheme functions around the primitives. Then a Scheme library can be created which provides a set of common functions based on the primitives. The next few sections describe the general functionality and interfaces that are implemented in the dtype cryptography library.

3.3.1 Random Number Generators

Random number generators (RNG) take a seed and then create random bits and bytes upon request. It is fairly easy to make a cryptographically secure random number generator, if you assume that you have some source for good randomness. For example, taking timings of random keypresses at a local keyboard is a relatively good source of randomness; watching network activity probably is not [3].

Because of the object-oriented nature of Scheme, the goal is to create cryptographic objects that maintain some internal state and perform some function. A random number generator object is created by specifying a generator function and a seed that is used to initialize the RNG. Once a generator is created, a user can pull off a random bit, a random byte, or a block of random bytes.

There are two random number generators that are currently supported. The first is a Linear Congruential (LC) RNG, which uses a 32 bit seed and does multiplications and additions to generate random numbers. This is fairly similar to what is done by the Unix random function. It is not very cryptographically secure, because it does not stir the random pool, but it is easy to setup and use.

The second RNG is called X917. It uses a user-defined block cipher to ensure the randomness of its output. The block cipher is used to stir the bits in the random pool to ensure a truly random output. This assures that it will generate a set of cryptographically strong random numbers. So long as there is sufficient randomness in the initial seeds given to the cipher and the RNG, the X917 RNG should result in a good set of random numbers. This cipher is created by specifying a block cipher algorithm, a key for the cipher, and a seed for the generator. In this case, the seed is the same size as the blocksize of the cipher (see section 3.3.3).

3.3.2 Hash Functions

A hash function takes a block of input and performs an algorithm to convert it into a fixed-sized block of bits that tag the input. The same input, when run through the same hash algorithm, will result in the same output. These algorithms are sometimes called one-way functions, because it is computationally infeasible to find two source files that hash to the same value.

To break a hash would require a large amount of mathematical work or trial and error. Changing a single bit in a input file will drastically affect the output of the hash function. Randomly changing the bits in a file would require 2^k attempts to duplicate the hash, where k is the number of bits in the hash's output. Using the birthday paradox, finding two random messages that hash to the same value would require $2^{\frac{k}{2}}$ attempts, on average. It is infeasible to break a hash because performing this many computations is practially impossible.

Two hash functions are currently supported, MD5 [15] and SHS [7]. A hash object is created by specifying which hash algorithm to use. Data is then fed into the hash object one piece at a time to update the object context. For example, if a very large dataset needs to be hashed, one could initialize a hash object and then update the hash in small increments as the large file is processed. At the end a hash is finalized, which means that it returns the value of the hash and resets itself to the initial state for use in a future hash.

3.3.3 Block Ciphers

Block ciphers are also called secret-key ciphers, and are what most people think about when cryptography is mentioned. A block cipher takes a block of text and encrypts it using some secret key. A recipient needs to use that same key in order to decipher the data into something meaningful, otherwise it just looks like random bits.

A block cipher object is instantiated by specifying the algorithm and key to use. This object can then be used to encrypt and decrypt blocks of data. One of the limitations of these ciphers is that it has a blocksize, and data must be encrypted in blocks of that particular size. For example, DES, the Data Encryption Standard, has a blocksize of 8 bytes. Therefore all DES encryptions and decryptions must be done in blocks of 8 bytes, even if only one byte of real data exists.

In order to evade this problem, the data is padded out to the first blocksize before it is encrypted, and after it is encrypted the actual size of the real data is prepended to the encrypted block. The real size is encoded as 4 bytes in MSB order. This results in a final block of data with is $\lceil datasize \rceil_{blocksize} + 4$ bytes in length, which is 4 bytes beyond the smallest blocksize boundary.

Currently, only two block ciphers are supported by the dtype crypto library. The first is DES [13], which has a blocksize of 8 bytes and a keysize of 8 bytes. The second is IDEA [10], which has a blocksize of 8 bytes and a keysize of 16 bytes. Since the user needs to know these data sizes, functions exist for users to determine these numbers for each algorithm supported. The blocksize is determined by querying a block cipher object once it is created. However the keysize needs to be known beforehand, so that can be queried by using the name of the cipher, just like when creating a block cipher object.

3.3.4 Public Key Ciphers

Public Key Cryptography is a relatively new concept, and puts a twist into the concept of secret codes. In a public-key cipher, each entity has two keys, a public key and a private key. The private key is kept very secure, such as on a floppy disk held all the time; the public key is released to the world, put in a white pages, and printed in the New York Times. These two keys are mathematically related to one another in such a way that encrypting a piece of data with one key requires the use of the other key to decrypt it.

The relationship between public-private keypairs is what enables the use of public key cryptography. For example, using someone's public key one can encrypt a message that only the owner of that key can read, since encrypting with the public key requires the private key to decrypt, and only the owner has that private key.

The reverse is also true. If a user encrypts a message with his private key, then anyone in the world can decrypt this message using his public key. Since only this one user could possibly encrypt something with the private key, it is known that this user must have sent this message. In effect, the user has *digitally signed* this message.

A public key object can encrypt and decrypt a message, as can a private key object. Also, since a public key is derived from a private key, a user can insert a private key in place of the public key operations. But the reverse is not true.

It is technically infeasible using today's technology and today's techniques to determine the private key from the public key. For example, the RSA cryptosystem is based upon factoring large composite numbers into two large prime factors. The largest number of this form to be factored is a 129 decimal-digit number called RSA-129 which was factored over an 8 month period using 1600 machines in 25 countries [2]. Normal usage of the RSA system employ keys in the range of 155-400 digits, which cannot be feasibly broken using the same technology that was used to break RSA-129.

In reality, public key ciphers are very slow compared to block ciphers or hash functions. In addition, most public key ciphers are limited in the size of the data that can be encrypted. As a result, they are usually combined with the other crytography functions to implement full crypto systems. For example, to encrypt a message it is encrypted first using a block cipher with a random key, and then that random key is encrypted using the public key cipher. Alternatively, a message is run through a hash function and the hash result is encrypted with a private key to generate a digital signature.

Currently, the only public key cipher implemented in dtypes is the RSA Cryptosystem [14]. In the future, other systems may be added. One possibility is the DSA algorithm [6] which is a standard put forward by the NSA to standardize digital signatures. Unlike RSA, the DSA algorithm only supports signatures, and does not allow for encryption.

3.4 Implementation

By combining the different ciphers and sets of Scheme code, a secure evaluator package was created which allows media bank servers to protect themselves from rogue clients. The `secure-eval` package provides a set of functions that servers can use to set up, initialize, and run a secure dtype server/evaluator. This section describes the implementation of this package. For the Scheme code that implements the secure evaluator, see Appendix B.

The major goal in implementing the secure evaluator was to provide a simple interface that could easily be applied to any dtype server. By supplying a simple interface, the job of modifying other software to use the secure evaluator is eased. In fact, a server can be changed to use the secure evaluator, in general, by changing a single function.

The secure evaluator starts by creating a server session and running a set of predefined initialization hooks. Once the server is running, it waits for client input. When a client connects, `secure-eval` will run a set of server-defined hooks which can initialize some server state for each client that connects. These hooks allow the server to maintain information about each client, such as open files or authorization information.

Once a client is connected, it can run commands on the server. In normal operation, when a data packet arrives from a client the server will evaluate the packet. This packet travels the network in clear-text. There is also an option to setup link encryption. If this option is set, then `secure-eval` will find the encryption state for the client and perform the appropriate decryption and verification on the incoming packet before it is evaluated.

When the evaluation has finished, the server will send a reply back to the client. If link encryption has been enabled for this client, the server will sign the reply packet and perform the appropriate encryption before sending the reply. In this manner, older clients can still use `secure-eval` servers, but clients that understand encryption can enable that option and secure the connection. Clients that do not use encryption are left with a weak link, which leaves the connection susceptible to network sniffing and network spoofing attacks.

Finally, the server can define a set of hooks to be performed when clients disconnect from the server. These hooks are useful for cleaning up after a client logs out. For example, the server can destroy security contexts, close and open files, or accumulate some accounting information.

In order to provide the security functions, `secure-eval` keeps a map of clients, and for each client it maintains a security parameter vector. This parameter can be set on a per-client basis, since this is what controls whether the evaluator will perform any encryption functions. The secure evaluator package provides a function to set the security parameter using public key cryptography as the key exchange method.

If a client wants to set up a secure connection, it generates a random session key, encrypts that key along with the block cipher name in the public key of the server and sends it across the network. The server then decrypts this message with its private key and sets the client's security parameter to reflect this key state. The client can change the current key at any time by executing the key-exchange function again. Alternatively, if the client wants to turn off security, running this function with a null token will turn off all security and return `secure-eval` to its original functionality, and susceptible to attack.

It is advised that the server provide a wrapper function around the `secure-eval` exported function to the client, rather than providing the original function itself, since the client should not have access to the server's private key. In the future, it would be better to provide a real-time key exchange method like Diffie-Hellman [5], so both the client and the server have some input in choosing the session key.

Chapter 4

Media Bank Transport Layer

The media bank transport layer is the means by which clients can transfer data to and from the media bank. Once the information about an object is obtained from the item server, a client uses the transport layer to actually download the data contained therein. The key design goal of the transport layer is to transmit object data efficiently and securely to the client.

4.1 Transport Layer Requirements

In theory, any means of transport can be used for the transport layer. However there are certain requirements that must be met for the transport layer to be useful. Each of these requirement confines the transport layer further, restricting its generality. Ultimately, a dedicated transport server will be necessary.

Minimally, the transport layer must take an object name, in some internal format, and return the object which is identified by the name. This requirement is straightforward. A client requests an object by name and the transport layer returns the object to the client. This could easily be accomplished using something like NFS, the Network File System, where a client can mount the server's filesystem and read the desired files directly from the server.

One drawback of this setup is that any possible media bank client needs to be able to mount every possible server. In addition, those servers become limited in where they can store objects, since anything on the exported file systems can be read by network clients.

The media bank requires that only defined objects be available. A server can choose which objects it will export, and where those objects should be located on that server. NFS does not fit this requirement, since anyone who could mount the filesystem could read anything on that filesystem, whether or not the server intended to export that data.

Next, the transport layer should provide for dynamic access control. This requirement means that the transport layer should be able to allow or deny clients download rights for objects in real time. For example, a client could be restricted to only download an object once. NFS could provide access control by utilizing user groups and using unix permissions to control the access to objects, but this is hard to change dynamically, and doesn't provide for one-time downloads.

A media bank server operator may wish to keep accounting information or usage statistics. Perhaps the goal is to discover the popular items, how often some objects are retrieved, or what combination of objects are retrieved together. The transport server should provide a mechanism to allow the provider to keep such data.

Another requirement of the transport layer is security. For example, only the client should be allowed to read the requested data, and network sniffer should be denied the ability to read the data at all. NFS cannot provide the level of security necessary to meet these requirements.

4.2 The Transport Server

Although NFS was useful for prototyping some of the media bank ideas, it does not fit all the requirements for a secure transport layer. As a result, a transport server has been created to take over the task of serving media bank objects. This server will only deliver data that is exported from the media bank (as opposed to NFS which will deliver any file exported). It can perform accounting, keep track of what items have been downloaded, and it can require dynamic authorization before it will deliver an object.

The media bank transport server is a dtype server which provides intelligent methods for object retrieval. A transport server resides on every machine on which objects are stored. Clients who wish to download objects from the media bank must talk to the transport server in order to obtain the data they require.

Because a media bank object can contain a sequence of discrete parts, methods are required to access the individual parts of an object. In the original design of the transport layer, it was believed that there were two methods by which a client could access data stored on the server. The first method is space, accessing the data as raw bits using a byte offset into the data and a length. The second method is time, accessing data through some time-based index and retrieving some quanta of data at that index. These methods are described herein.

4.2.1 Retrieval in Space

The easiest method for a client to access data is to extend the idea of the file system over the network. The idea behind retrieval in space is based on that of a limited network filesystem. A client connects to the transport server, opens the appropriate object, and then reads and seeks into the file at the desired offset, as if it were in a normal file system. Then the client requests the appropriate number of bytes from the server.

Using this method to obtain data from the media bank servers, a client needs to know something about the data it is retrieving. It needs to know about the format, content, and methods of the stored data in order to access it properly. While it may be acceptable to require some clients to have this knowledge, it is important to also allow less intelligent clients access to data.

4.2.2 Retrieval in Time

To access data in time, a client requests a time-offset into the data set and retrieves an appropriate-sized block of data from the server. The idea behind this retrieval method was to abstract out the time dependencies of multiple encoding methods, so the client doesn't need to know how the data is encoded.

For example, take a file containing a sequence of JPEG encoded images at some regular frame rate. This file would have the property that not every image is the same size, so it is not possible to just seek to some place in the file to find a specific data offset. If the client wanted to find frame 1995, or the frame at time-index 66.50 (knowing the frame-rate of 30 frames per second), it would need to either read the whole file from the start, or know something about how the file is stored on the server's disk. Clearly this is too cumbersome for this type of data.

If, however, the server was able to perform those functions for the client, either by knowing the on-disk format of the data, or perhaps having a file containing hints about the on-disk format, then the client would only need to ask the server for some known time index to retrieve the data. This is the idea behind the time-domain access to media bank data. The client does not need to know where to look to find data, since the server can perform the offset and size lookups on its own.

4.3 Transport Server Implementation

The transport server is implemented as a dtype server based upon the secure evaluator (see chapter 3). This means that the transport server is using a Scheme interpreter, but only provides a limited interface to its clients. This limited interface is for security reasons (see section 4.4) so that clients do not get a full run of the machine running a transport server. For a full explanation of the transport server functions and the code that implements it, see Appendix C.

The basis for the transport server implementation is that of a network file server. A client sends a token to the server and requests a media bank object. The server validates the token to make sure the client is allowed to download the object. If the validation succeeds, the server will open the datafile associated with the object and generate a random handle to send to the client. This handle binds the datafile to the client and provides some state in the server, which maintains a map of handles for each client.

When a client wants to access the data in an object, it requests the data, in time or space, from the server and provides the handle that was returned previously. The server uses this handle to find the object and processes the request. Because the handle is stored along with the client state, the object handle will only be useful to the client that opened the object. In this manner, a network snooper cannot see an object handle and then attempt to use it, since the attacker will have a different state on the server so the snooped handle will not be valid.

Once the client is finished with an object, it contacts the transport server which closes the datafile. This way the server can maintain a list of open files, and if a client exits without releasing its resources, then the server will close them automatically.

Since the transport server is using the secure evaluator, a client can choose to encrypt the network session. The transport server supplies the client with a hook to initialize a key exchange protocol, and also provides each client with the opportunity to download the server's public RSA key.

4.4 Transport Server Security

There are four aspects of security which are relevant to the media bank. These aspects are data integrity, data confidentiality, user authentication, and user authorization. These aspects are described in following sections.

In addition, the transport server provides a limited interface to the client. If the client were to have unlimited access to the server, an attacker could theoretically redefine some of the server functions to bypass any form of security. Consequently, a limited interface is provided to the client so that security can be maintained.

4.4.1 Data Integrity

Data integrity is protecting the transmitted data from change. While the contents of the data might still be readable via a network snooper, an active attacker cannot change the data en route. A user would like to ensure data integrity to guarantee that any data that was transmitted makes it to the other side intac. The recipient would, at the very least, like to know if the data were altered.

There are a number of ways to achieve data integrity, but it usually involves adding a digital signature, in some form or another, on the data. This digital signature can either be a public-key or a private-key signature, since the actual sender of the data is not needed by an outside party. As a result, it is possible to create a data integrity protocol using only secret-key cryptography; the two parties only need to exchange a secret key at the onset of the communication.

One problem with this scheme is a denial of service attack. Someone could modify the data mid-stream, such that the receiver can no longer decode the data properly. In this case, the server would not respond to a request, or the client would not receive the appropriate data. Some data integrity systems provide mechanisms to protect against these attacks, using error correction codes. The current implementation does not protect against this attack.

4.4.2 Data Confidentiality

The term *confidentiality* refers to the actual protection of the data being sent and received. The idea is that only the sender and the recipient can know what the data is, and any attacker who is listening to the network can only read random data.

Data confidentiality is provided by encrypting the data in a secret key known to both parties. Because confidentiality does not suffice for integrity, and vice-versa, the two operations are called orthogonal. I.e., a connection can have integrity, confidentiality, both, or neither.

4.4.3 User Authentication

User authentication is a proof of identity. It is a means for a server to be sure who its client is. For example, MIT has a system called Kerberos [9, 16] which authenticates users to services and services to users using a trusted third party, the kerberos server, which delivers tokens to its users. Using a cryptographic protocol, a server can be sure that a user is who he claims to be by reading the token. Since the trusted server vouched for his identity, the token would prove the user's identity.

In the media bank, a user's identity does not need to be known by the server, although it would be advantageous for the server to authenticate to the client. Therefore, user authentication is not used, although server authentication may be used.

4.4.4 User Authorization

Authorization is similar to authentication, except instead of vouching for a user's identity, the token passed from client to server vouches for the client's ability to perform some function or operation. For example, the authorization token could tell the server that this user has the ability to read some piece of data, or change some piece of information.

In the media bank, the authorization system is a digital voucher system, based upon the concept of a *digital movie ticket*, as described in chapter 5. A user presents the authorization token to the transport server when requesting data to prove the ability to obtain that data.

Chapter 5

Digital Movie Ticket System

The digital movie ticket system is used to authorize clients to download data from the media bank. The idea behind the digital movie ticket system is that a client can purchase a program from the media bank once, and then be allowed to retrieve all the objects that comprise that program. In this manner, the client only needs to obtain a single authorization token, or voucher, which can be used to download an entire program. The system is designed to have a central clearinghouse for the tickets, allowing media bank servers to verify the authenticity and validity of tickets before rendering services. However the media bank servers should be able to verify the authenticity of the movie ticket itself and only require the clearinghouse if it wants to enforce one-time downloads; servers can opt to perform expiration-time validation instead.

5.1 Digital Cash

Digital cash, or digicash, is an electronic equivalent to anonymous money. In its simplest form, the digicash protocol starts with a bank which issues a token to a client. The client can use that token with a vendor, who in turn redeems the token with the bank. In order to act like normal cash, digicash requires that tokens can only be spent once. Moreover digicash, like real cash, has no binding between the token and the user's identity [1, 8, 12].

For example, if Bob wants to go to a pizza truck and buy a slice of pizza for $1.00, he just walks on up and hands over the coin (or bill, as the case may be). Similarly, if Alice wants to buy a piece of data, she only needs to turn over the digicash token. Just as the pizza truck doesn't need to know who Bob is, the data vendor doesn't need to know who Alice is. Just like cash, the online token doesn't contain anything to identify Alice.

In general, digicash can be broken into two categories, online and offline verification. Offline verification is where the client and the vendor perform their transaction at one point in time, and then the vendor redeems the digicash token at a later time. In general, offline digicash either requires a trusted observer, or enough extra bandwidth to expose a user when they double-spend the currency.

Online digital cash, however, requires the vendor to contact the bank in the middle of the transaction to verify the digicash token. The problem with online digicash is that as the number of clients increases, the load on the banks becomes hard to bear because of the protocols involved. These protocols are discussed in the next chapter.

Because the bank needs to verify every transaction, as the number of transactions increases, so does the burden put on the bank's server. The ideas of the digital movie ticket protocol are based upon online digicash. Since the media bank is inherently an online system, adding another online server to the system does not add a significant burden to the whole bank. By offloading the overhead of validating tickets, the transport server can save some time performing the cryptographic operations, which can be time-consuming.

5.2 Online Digicash Protocol

Digicash is based upon the belief that in an integral field, exponentiating numbers is easy, but taking roots, in general, is hard. So, for example, it is easy to perform the function

$$x^3 mod N$$

however it is difficult to perform

$$x^{\frac{1}{3}} mod N$$

without knowing the factorization of the field size N. The algorithm described below is only one form of digital cash. Many other algorithms exist, however only the online system is relevant to the topics in this thesis.

5.2.1 Digital Signature

If you assume that some entity, like a bank, does know the factors of N, then only that entity can easily take cube roots $mod N$. The result is that anyone who sees a number of the form

$$x^{\frac{1}{3}} mod N$$

can 1) verify that it is of this form, and 2) know that the entity must have performed the initial operation. Digicash works by having the client present a coin ID and a token

$$x, x^{\frac{1}{3}} mod N$$

to the vendor, which can verify the bank's digital signature on that token. The signature is verified by decrypting the token and comparing it to the value of the coin ID x that is given by the client.

Looking closely, one can see the hole in this statement. Since anyone can perform exponentiation, it would be easy for someone to let

$$x' = x^3 modN$$

and then present x' as the cube root of x, thereby fooling a server to believe that this is valid digicash. In other words, a client could generate a digicash token on the fly by sending the digicash coin

$$x', x$$

without the need for the bank, and vendors would not be able to differentiate from a real token.

5.2.2 Adding Secure Hashes to Prevent Client Spoofing

To protect against rogue clients generating tokens, the protocol uses a secure message hash, like MD-5 [15], Ron Rivest's Message Digest 5, to generate random values. The protocol uses the hash to scramble the coin id before sending it to the bank. So, the client generates a random coin ID, x, computes its hash, $md5(x)$, gives the bank the value of the hash, and gets back this value[1]

$$md5(x)^{\frac{1}{3}} modN$$

Now, when the client wants to use the coin, he gives the digicash token $(md5(x)^{\frac{1}{3}})$ to the vendor, who verifies with the bank that it has not been used, and then the client gives the vendor x, which can be used to verify the token.

Because hash functions are one-way, it is computationally infeasible to compute x given $md5(x)$. So, a client cannot exponentiate the md5 herself, since computing the inverse hash is impossible. Hence, that attack is thwarted.

[1]For brevity, the $modN$ is implied in all further computations of this sort

The only problem with this scheme is that the bank can follow the digicash token from client to vendor. Since the bank sees $md5(x)$ when it signs the token, and again when it redeems it, the bank now has a path from the client to the vendor for this token, and can keep tabs on the client. Since one of the requirements for digital cash is anonymity, this clearly will not work.

5.2.3 Blinding the Token

The solution is to *blind* the token before the bank sees it. Recalling that exponentiation is easy, the client can create a blinded token by generating a random coin ID x, and a random blinding factor, b, and sending the bank

$$md5(x)b^3$$

The bank doesn't know b, so it can not discover $md5(x)$ from this value. The bank signs this value, and returns

$$(md5(x)b^3)^{\frac{1}{3}} = md5(x)^{\frac{1}{3}}b$$

The client can divide out b, and now has the digicash token,

$$md5(x)^{\frac{1}{3}}$$

as above. The client cannot use the blinded value as a digicash token, since the client theoretically cannot find a y such that $md5(y)$ equals $md5(x)b^{\frac{1}{3}}$.

When the redemption of this token happens, the bank has nothing with which to match it, so cannot match the vendor redemption to the client request. The client's anonymity has been preserved, since the bank cannot track the coin from vendor to client. Coins cannot be double spent, since the vendor verifies with the bank that the token has not been seen before. And clients cannot generate tokens themselves, since they cannot reverse MD5. This is the protocol used by many online digicash schemes today.

The money is substracted from the client's account when the bank calculates $(md5(x)b^3)^{\frac{1}{3}}$. The cash is transfered to the vendor when the bank verifies the token. The client thereby loses the use of the token, and subsequent uses be flagged as double spending.

5.3 Digital Movie Ticket Trials

Theoretically, the media bank could use plain digicash to pay for its objects. The problem with this is that some media bank programs may be made of hundreds, even thousands, of individual objects. If a client had to pay digicash for each and every object, this would both overwhelm the bank as well as the media bank transport servers, not to mention require very small values of digicash, which must be in integral, quantized values.

One solution is to meter the usage and charge the client at a later time. A problem with this solution is that it becomes impossible to know if a client already obtained some object. If the system wants to enforce one-time downloads, a client either has the opportunity to obtain data twice, or gets charged for the same data twice. Moreover, this solution requires a protocol to obtain payment after the data transfer, or if pre-payment is used a means to refund the unused portion.

Another solution is to take generalized digicash, and put a limiting factor on it in the form of an authorization field. This authorization field is used to tell the media bank servers what program the client bought. The field is also called a *pag* for process authorization group. Because of the pag, a media bank transport server only needs to validate the ticket and compare the pag to the valid authorization fields for an object.

One requirement for the authorization field is that both the client and the bank must be able to see and verify the pag before service is rendered. The bank needs to know the pag before it signs the movie ticket, to validate the user's purchase of that program. The client needs to see that the proper program is being bought. In other words, both the client and the bank need to know the pag and prove that the pag is in the token.

Another requirement of the digital movie ticket is that it should be anonymous. The identity of the user should not be in the ticket, and there should not need to be any correlation between the ticket and the user. The movie ticket server should not be able to track any single token from client to vendor, nor should it be able to correlate multiple tickets for the same, or any, client for any purpose. To accomplish this end, a number of schemes were explored. Each different trial is discussed below, as well as an analysis of their failure modes.

5.3.1 CoinID and PAG in MD5

This trial is just an extension of the digicash protocol presented at the end of section 5.2. The idea is to include the pag in the MD5 calculation with the coin ID. In this trial, the client would send

$$md5(x, pag)b^3$$

to the bank, and after unblinding would have the value

$$md5(x, pag)^{\frac{1}{3}}$$

The problem with this method is that there is no way for the bank to verify the pag before signing the token. The bank doesn't know b, so can't find $md5(x, pag)$, and doesn't know x, so couldn't find the pag even if it did know b. The consequence is that the client can insert any pag desired, and the bank would be none the wiser.

5.3.2 Send PAG separately

Instead of including the pag in the md5 calculation, another possibility is to send a normal digicash token to the server and include the pag alongside. I.e., a client sends

$$md5(x)b^3, pag$$

to the server, and the bank sends back

$$(md5(x)b^3 md5(pag))^{\frac{1}{3}}$$

which unblinds to

$$md5(x)^{\frac{1}{3}} md5(pag)^{\frac{1}{3}}$$

This solves the previous problem of the server not knowing the pag requested by the client. Both the client and the bank can see the pag, and the vendor and bank can verify this token given both x and the pag.

The problem with this system is that the client, knowing x and pag, can separate the token into its two parts,

$$md5(x)^{\frac{1}{3}}$$

and

$$md5(pag)^{\frac{1}{3}}$$

and then mix-and-match different pairs of values. For example, a client could purchase an expensive program once, save off the signed pag for that program, and then in the future purchase a very cheap program and insert the expensive pag.

5.3.3 Send PAG in Blinded info

This method tries to include the pag in the blinded token by sending it as an exponent to another value. In this method, the client creates the token ID, x and generates another value, y, which is relatively prime to $md5(x)$. Putting it together, the client sends

$$md5(x)y^{md5(pag)}b^3$$

and then gets the return value from the bank, which unblinds to the value

$$md5(x)^{\frac{1}{3}}y^{\frac{md5(pag)}{3}}$$

which the client can attempt to use as the ticket.

The problem with this is twofold. First, the server doesn't know the pag value beforehand, and even if it did, there is no way for it to verify the pag. Second, the client could again separate the pag part from the token ID, and create a new pag.

The first problem can be solved by the client sending y and the pag to the server with the request:

$$y, pag, md5(x)b^3$$

and then have the server do the multiplication. But even using this method, the second problem will remain; the client will still be able to pull apart the token into the ID and pag parts.

5.3.4 Trial Conclusions

In all the trials to design a digital movie ticket, it seemed impossible to meet all the requirements of the system. In particular, either the server had no way to verify the ticket, or a rogue client could manipulate the result to generate any ticket wanted. Neither of these results are useful, however they lead to the conclusion that the constraints on the system are too tight to allow a solution at this time.

Given the current state-of-the-art, the best solution is to relax one constraint on the digital movie ticket, anonymity. If we allow the entity that creates the digital movie tickets to follow that token from client to server and back, if we allow it to trace the token, then we open up the possibilities for elegant solutions to the rest of the problem.

Losing this little bit of anonymity is an acceptable loss. There is not a necessary binding between the client and any one individual, so the server creating the digital movie tickets need not require a real identification, just real payment. Consequently, it remains possible to retail inter-token anonymity; the movie ticket server does not retain the ability to match the identity of any two voucher-holders.

The result is that a movie ticket server has the ability to track a single ticket through the system, and can obtain the list of purchases done using that one token. However the server cannot correlate two different tokens to find out if they were used by same client.

5.4 Digital Movie Tickets

The payment system for the media bank is a digital movie ticket. A client purchases a movie ticket from a movie ticket server, and can present that ticket to the specific vendors when obtaining the selection. For example, a client who wishes to view the movie *T2* from the media bank would purchase a *T2* movie ticket, and then present that ticket to the specific media bank servers upon which the *T2* movie parts reside; those servers, or vendors, then contact the movie ticket server to verify the movie ticket.

5.4.1 Digital Movie Ticket Design

A digital movie ticket is a digitally signed dtype (see section 2.2). The movie ticket server has an RSA key [14] which is uses to digitally sign movie tickets. The movie ticket object is converted to a byte-stream, which is hashed using md5, and then

tokenid	This is a unique id for this movie ticket token.
now	This is the current date/time in UNIX-time, the number of seconds since 00:00 Jan 1, 1970 GMT.
expires	This is the UNIX-time when this token expires.
pag	This is the authorization token which describes what this token will do. In many cases, this could be an object name, or a list of objects, or a program name, or some other dtype describing what the user wants.

Table 5.1: The required fields for the digital movie ticket

encrypted using the movie ticket server's RSA key. A client or a vendor can verify the ticket by converting it to a byte-stream, running md5 on it, and comparing that to the RSA-decrypted signature.

By using dtypes as the basis for the movie ticket, upgrading the tickets in the future is easy; just change the data in the object. Another feature of this design is that the protocol becomes simple, since we already have the means to transmit dtypes. The only requirement of the system is that the same dtype object be converted to the same packet format each time it is used. The movie ticket format is a vector of keyword-value pairs. There are a limited number of required fields, listed in table 5.1.

Because the token is a dtype object, any number of additional fields can be added at a later time. Optional fields are also allowed. The only requirement, as already stated, is that the conversion to a byte-stream remain constant.

5.4.2 Digital Movie Ticket Protocol

The movie ticket protocol is rather simple. A client connects to the movie ticket server and requests the selection. Along with the request, the client sends the payment for the program. The server creates the movie ticket, signs it, and returns it to the client.

Since the ticket is a dtype object, it is possible to use the existing network transport protocol to transmit the dtypes across the network. The only thing that needs to be protected is the network traffic, so someone listening to the network doesn't obtain the movie ticket.

This connection is secured using some public-key technology to coordinate a key to encrypt the session. Most likely, Diffie-Hellman key exchange will be used to choose a DES or IDEA key to secure the connection, although some variant of RSA could also be used to perform the key exchange. In this manner, the client and server can perform some protocol to choose an encryption key which will be used to secure the session.

Once the client has the movie ticket, she can present the ticket to any of the vendors which contain those objects. Those vendors verify the movie ticket, and allow the client to download the object if the movie ticket is valid. See figure 5-1 for a representation of the communication paths.

5.4.3 Movie Ticket Server Implementation

The movie ticket server is based upon the secure evaluator package described in chapter 3. The server waits for clients to connect, and expects clients to use the encryption methods of the secure evaluator to protect the connection. It accepts some form of payment, such as digital cash, and returns a movie ticket for the appropriate selection back to the client.

The movie ticket server needs to be connected, in some manner, to the directory server (see section 2.3.1). When a client asks for a media bank program by its name, the movie ticket server needs to find out which objects comprise that program. By asking the directory server, the movie ticket server has the opportunity the find all the objects in the media bank which the client wanted and put them into the movie ticket.

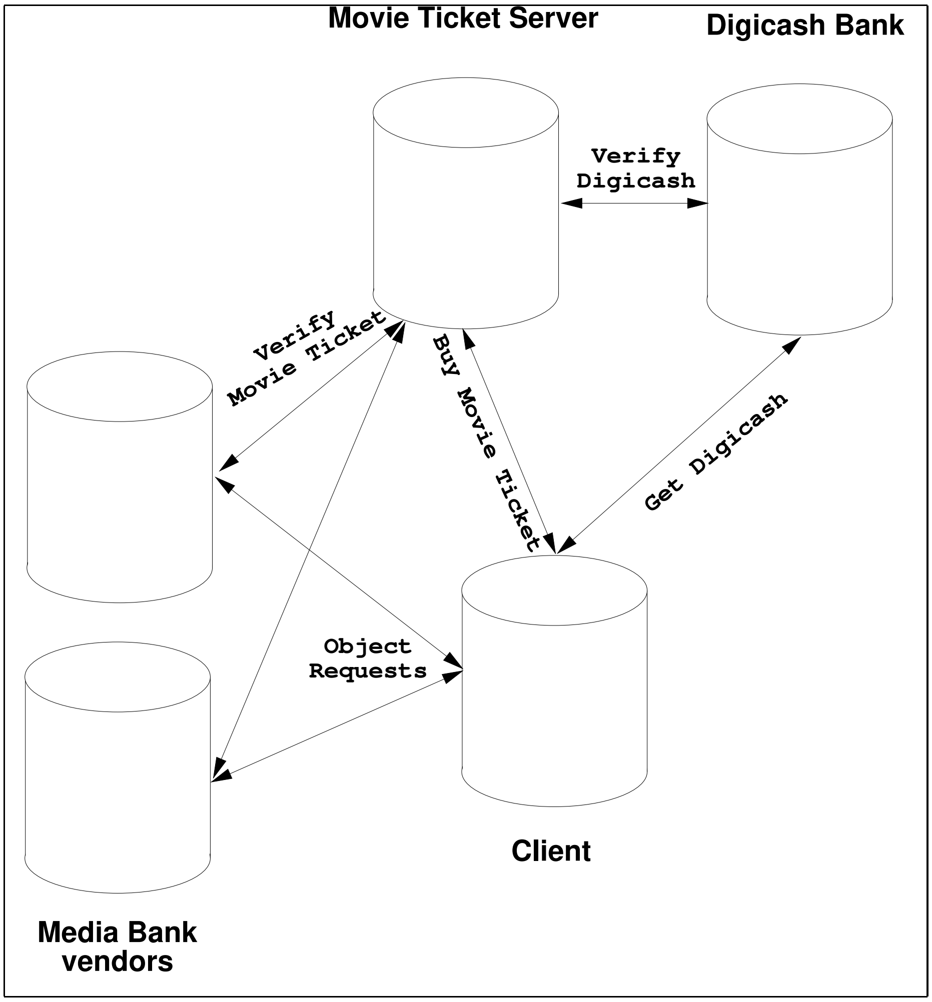

Figure 5-1: A visual depiction of the communications between the client, movie ticket server, digicash bank, and media bank servers.

Once the list of objects has been obtained, a movie ticket is generated for the client. The ticket contains the list of objects, the current time, an expiration time, a token id, and the signature of the movie ticket server to validate the token. The signature is made using the server's RSA keypair, signing all the data in the ticket.

A transport server takes the token from the client and checks its authenticity by first verifying the signature on the token, and then checking the validity period. Next it compares the object request with the token, to see if the requested object is authorized by the token. Finally, it will return the object if the client is authorized.

Chapter 6

Conclusion

The security services described herein will allow the media bank system to be used in a commercial environment. Service providers can be assured of both the security of their servers and payment for their services. Therefore, the goals of the media bank security services have been met.

One goal of the media bank security services was to allow dtype servers to be run on a machine without compromising the security of the server or the security of the data. This was accomplished by multiple means: the Scheme evaluator in dtypes was secured, the data stream was encrypted, and the data was protected by an authorization system.

First the dtype evaluator was secured. Originally, any client that connected to a dtype server could have full run of the Scheme evaluator, which resulted in a full run of the machine. By limiting the access to a set of server-defined functions, clients no longer have full run of the server, so security can be maintained. A carefully designed server can limit the client to only perform functions that the programmer allows.

Second, the network transmissions were secured. A number of cryptographic functions are used by the client and server to setup a secure communication channel across the network which could be used to send the data. This channel can be negotiated to require digitally signed and/or digitally sealed objects. In this manner data confidentiality and data integrity is ensured since only the two parties can perform the required operations to successfully sign and/or seal the data.

Finally, the data on the servers was restricted using an authorization system, the digital movie ticket. Clients obtain a ticket for the appropriate selection of objects, and then present the ticket to the transport servers to prove that he is authorized to obtain the data. A single ticket is used for the whole media bank selection, to reduce the amount of data that needs to be exchanged. The server can choose how to honor the ticket.

6.1 Future Work

There exist significant opportunities for additional enhancements to the system: different cryptographic systems can be adopted, alternative key exchange protocols can be implemented, performance tests can be run, and so forth.

There are many cryptographic systems available today, and more are becoming available as time goes on. The choices made for the supported dtype cryptographic systems was to reduce implementation overhead; the algorithms were chosen because code existed to implement them, and their behavior was well known. However there are other algorithms that exist today which should probably be added to the repertoire of functions, such as stream ciphers or triple ciphers, which add speed and security to the system.

In addition to the bare-bones crypto systems employed, other means of exchanging keys and performing cryptographic handshakes will be needed. For example, an implementation of Diffie-Hellman Key Exchange would improve the key exchange protocol for both clients and servers. It would allow a real-time key exchange which allows both client and server to have a say in the final key, rather than one side dictating the key to be used.

Another problem that needs to be explored is testing the performance of the servers. The dtype system was not written with real-time systems in mind. It would be a good idea to measure the performance of the servers to see if they can handle the amount of data required within the time constraints. Right now data can be buffered by the client, but it would be better if very little buffering were needed.

It is unclear if the dtype system can perform under high load. A single client works, but running five clients has not been tested. Since no one has tested the media bank under any load, it is unknown how the servers will react.

Moreover, no one has tested the media bank servers with large databases of objects, or with many item servers. The prototype system only has a handful of item servers with a total of only a few hundred objects. It is unclear how the current system will react to the additional load of many servers and a multitude of additional objects. Clearly this should be tested at some point in the future as well.

Auditing trails are useful in a commercial environment. They provide information about usage patterns which can help marketers and implementors alike. For example, it would be useful to know which selections in the media bank are downloaded most often, or whether some scenes are more popular than others in some movie, or which servers get the largest number of requests. Currently, this information is not maintained anywhere, but it would be a simple task to add the code to retain this information.

6.2 Acknowledgements

I would like to acknowledge everyone who has helped with this project and this thesis in the last two years. First, thanks must be given to my advisor, Andrew Lippman, whose guidance has helped direct me and the course of my research over the last two years. I would like to thank my readers, Kenneth Haase and Daniel Schutzer, whose comments and suggestions have given me much food for thought over the last six months. I am also indebted to many people at the Media Lab who have either helped me personally or helped with the media bank project in general. These people include, but are not limited to, Walter Bender, Michael Best, V. Michael Bove Jr., Pascal Chesnais, Klee Dienes, Daniel Gruhl, Henry Holtzman, Frank Kao, Jill Kliger, Roger Kermode, Michelle McDonald, John Watlington. I would also like to thank all my friends who helped me with my writing. This project could not have happened without the generous sponsors of the Media Lab who have funded the Media Bank Project. I would also like to thank my parents, who have always encouraged me to strive to be the best I could. Finally, I would like to thank Heather Harrison, who has spent many hours awake both proofreading this document and waiting for me to finish working.

Appendix A

Cryptographic Dtype Functions

The following sections detail the dtype cryptographic functions. The primitive functions are documented in sections devoted to the type of cryptography. Following those will be a section detailing the supplied composite functions that use these primitives.

A.1 Random Number Generators

This section discusses the Random Number Generator. An RNG is an object that is created with some seed outputs pseudo-random numbers. Given the same initial state, two RNG objects will output the same set of random numbers. Therefore, a good random source is needed for the seeds.

Two RNG algorithms are supported, a Linear Congruential RNG, and an X917 RNG. The former algorithm takes just a numeric seed and performs addition and multiplication, linear functions, to generate random numbers. This algorithm is fast and easy to initialize, but its numbers are not very random.

The second type of RNG is the X917 RNG. This version takes an argument of a block cipher and a seed. The block cipher is discussed in a later section. The seed is a block of data at least as big as the blocksize of the cipher that is used. For more information on these terms, see section A.3.

`rng-create (type [cipher] seed)`: This function creates an RNG object. To initialize an LC RNG, the type is `'lc` and the seed is the numeric value. To initialize an X917 RNG, the type is `'x917`, the cipher is a block cipher object, and the seed is a packet of data at least as big as the block cipher blocksize.

`rng-get-bit (rng)`: This function will return a boolean value, true or false, depending on the parity of the next byte of random data.

`rng-get-byte (rng)`: This function will return a single byte of random data. In other words, it will return a number from 0-255 in the form of an integer.

`rng-get-block (rng size)`: This function will create a block of size `size` bytes of random data. This random data will be returned in the form of a packet.

`rng-noise-source ()`: This function will generate a 32-bit random word. This word is useful for as a seed to the LC RNG. This function uses a number of random sources on a machine, such as the current machine load and function usage, as well as some not-so-random information like the current time and process ID, to generate 4 random bytes of information. This information is returned in the form of a number.

A.2 Hash Functions

A hash object is a one-way function that takes input data and creates an output value based upon the input. Hash functions are deterministic. If the same input is given to the same hash function, the same output will be created. Two hash functions are supported, MD5 and SHS.

`hash-create (type)`: Creates a hash object of type `type`. Uses the MD5 algorithm if type is `'md5`; uses the SHS algorithm if the type is `'shs`. A created hash object is returned with its context initialized to the hash starting state.

`hash-update! (hash data)`: Updates the hash context with the data input. This function changes the context of the supplied hash object, updating it with the hashed packet of data.

`hash-finalize! (hash)`: Returns a block of data that reflects the current state of the hash context. When the hash is returned, the context is re-initialized to the starting state.

`hash-hashsize (hash)`: Returns the size of the data block that `hash-finalize!` will return for the hash. This way a user can know how much data to expect to be returned from a hash function.

A.3 Block Ciphers

A block cipher takes a block of data and encrypts it in a key such that the original text can only be obtained by using the same key. Different block ciphers have different key lengths and different block sizes. The key length is the size of the key used to encrypt and decrypt messages. A block size is the amount of data that needs to be encrypted and decrypted at one time. Currently there are two supported block cipher algorithms, DES and IDEA.

`block-create (type key)`: Creates a block cipher object. The object will use the `type` algorithm argument with the key `key`. For the DES algorithm the type is `'des`, and for the IDEA algorithm the type is `'idea`. The key must be the appropriate length for the algorithm type.

`block-blocksize (cipher)`: Returns the blocksize of the cipher object. This will return an integral value.

`block-keysize (type)`: Returns an integral value of the size of the key for the type of algorithm requested. The `type` is the same as that used in the create method.

`block-encrypt (cipher data)`: Encrypt the `data` with the `cipher`. The encrypted data will be in multiples of the blocksize. For example, if the blocksize is 8 bytes, then the encrypted data will be $8n$ bytes where n is the smallest value in which the data will fit in $8n$. E.g., if the data is 7 bytes and the blocksize is 8, then $n = 1$ and the encrypted data is 8 bytes. This function will return the encrypted data prepended with a four-byte datalength field used to recover the original data. The data-length field is a 4-byte MSB-first value that is the length of the original data.

`block-decrypt (cipher data)`: This function takes a block of data in the format output by the encryption function and decrypts it, and returns the plain text.

A.4 Public Key Ciphers

Public Key Ciphers involve two objects, a public key and a private key. Because of the mathematics involved in most public key systems, a private key can also be used as a public key, but the reverse is not true. Therefore, whenever a public key is required for a function, a private key can be supplied instead without problems.

Currently, only the RSA cryptosystem is implemented. However, the functions are supposed to be abstract enough to allow for multiple public key ciphers in the future. Some of them, however, are based upon RSA only, and will change when other systems are supported. The `type` of the RSA cryptosystem is `'rsa`.

`pubkey-create (type n e)`: Create a public key from the modulus and encryption exponent. The arguments `n` and `e` are only valid for the RSA cryptosystem.

Other systems will require other arguments.

`privkey-create (type n e d p q dp dq u)`: Create a private key from the modulus, encryption exponent, decryption exponent, the two primes p and q, and some intermediate values. When other crypto systems are supported, the arguments will vary depending on what the system requires of its private key.

`pubkey-create-from-iostream (type stream)`: This function will create a public key of type `type`, reading the the data from the iostream passed in. The data is expected to be in a format of BER-Encoded hex digits.

`privkey-create-from-iostream (type stream)`: This function will create a private key of type `type`, reading the data from the supplied iostream. The data is expected to be BER-encoded in hex digits.

`privkey-random-key (type rng keybits)`: This function will generate a random private key of type `type` with `keybits` bits, using the supplied random number generator as the source of bits. This function might take a while to compute the necessary values.

`pubkey-write-to-iostream (pubkey stream)`: This function will write the public key out to the stream in DER-encoded format using hex digits. This output is in proper format to be read in at a later time by `pubkey-create-from-iostream`.

`privkey-write-to-iostream (privkey stream)`: This function will write the private key out to the supplied stream in DER-encoded format. The output will be hex digits suitable to be read in at a later date. Since the output is a real private key, user's of this function should take great care in the output this function creates. Anyone who obtains this output now has the private key.

`pubkey-encrypt (pubkey data [rng])`: This function will encrypt a piece of data with a public key, using the pkcs encoding format. The data will be padded with random data obtained from the optional random number generator. If an rng is not supplied, the function will obtain random numbers using the LC rng using the noise source as a seed. This function is used to encrypt a message to a particular recipient; only the entity with access to the private key can decrypt the output of this function.

`pubkey-decrypt (pubkey data)`: In order to verify a signature, this function must be used. This will decrypt data that was encrypted in the private key and return the decrypted value. It expects data to be in the pkcs encoding format, like that output from the associated encryption function.

`privkey-encrypt (privkey data)`: This function is used to sign a message. The message is encoded using pkcs format and encrypted in the private key supplied. The result of this function is a block of data suitably used as a digital signature.

`privkey-decrypt (privkey data)`: This function decodes a message that was encrypted for this key. The data is expected to be in pkcs format. The decrypted value is returned to the caller.

`pubkey-extract (pubkey)`: This function returns a list of values that comprise the public key. In terms of RSA, this function will return the values of n and e.

A.5 Composite Functions

These procedures are supplied by the cryptographic library to the Scheme evaluator as a part of the `crypto` package. These functions are written in Scheme and are meant to add value to the primitives by providing a consistent interface to many useful services.

`make-signature (privkey data)`: This function is used to provide a digital signature on an arbitrarily-sized piece of data. The data is hashed using MD5 and then the MD5 hash is encrypted in the private key supplied. The resultant private-key-encrypted MD5 hash is then returned to the caller.

`verify-signature (pubkey data sig)`: This function is the opposite of the previous function; it verifies a digital signature. This function will compute the MD5 hash of the data, and decrypt `sig` with the supplied public key. If the decrypted value equals the computed hash, then the procedure returns true. Otherwise it returns false.

`seal-packet (data crypto-block)`: When one party wants to seal a message for another party, this function is used. The `data` is encrypted using the rules and methods contained in the `crypto-block` argument. The crypto-block should be a state that was agreed upon by both parties using some key exchange method. The output is a packet suitable to decryption by the next function. This function can sign a packet and/or encrypt the data. Signing is done first, then encrypting.

`unseal-packet (data crypto-block)`: This function is used to decrypt and verify a packet sealed with the previous function. The `crypto-block` is an associated array of crypto functions agreed upon between the two parties involved in the association. If the association calls for encryption, then the data packet will first be decrypted before anything else is checked. Next, if the data has a digital signature and there is a signature association, the signature will be checked and the data or an error returned. If there is no association, any signature on the data will be ignored and the data will be returned without error.

`setup-key-exchange (pubkey cipher key)`: This function will generate a key exchange initialization message using the `cipher` type with the key `key`, and encrypting the request to the public key `pubkey`.

`read-key-exchange (privkey data)`: This function decrypts a key exchange request and returns data suitable to create a crypto-block. In the future, the key-exchange functions should probably create the crypto-block itself.

Appendix B

Secure Evaluator Code

B.1 Exported Functions

The following paragraphs define the functions available to clients of the secure evaluator. These are the `secure-eval` functions that a Scheme server must use in order to utilize the functionality of the secure evaluator.

`secure-eval-add-connect-hook (f)`: A server that wants to run some function when a client connects should use this procedure to define a connect hook. When a client connects, `secure-eval` will run all of the defined connect hooks before allowing the client to run any commands. The argument is a function that should take two arguments, the server and the client: `f (server client)`.

`secure-eval-add-disconnect-hook (f)`: This procedure is used to add a function that is called whenever a client disconnects from the server. The argument, `f`, is a function that takes two arguments, a server and a client. The disconnect hooks are used to clean up resources that are used while a client is connected or resources that are initialized when the client connected.

`secure-eval-add-init-hook (f)`: This function is used to set up procedures that run at server initialization time. The argument `f` takes a single argument, which is the server object. When `secure-eval` creates the server object, it will run all the init hooks that have been supplied.

`secure-eval-current-client ()`: This function returns the current client object. This is useful for code that wants to find what client is actually calling the procedure, in order to make per-client decisions.

`secure-eval-set-client-security (token private-key)`: This procedure provides a means to initialize per-client encryption. The server is expected to create an intermediate function that the client can execute, which sits between the client and this `secure-eval` procedure. The client sends a key exchange request to the server, which then executes this command in the client's context. The server passes the client's key exchange token and the server's private key to try to decrypt the request and set the security parameters. If the token is null, then encryption is turned off.

`secure-eval-loop (port environment)`: This is the server-loop function for the secure evaluator. Once a server has performed all the initialization it wants, it then calls this function. The port number is the Unix TCP port on which the server expects to run, and `secure-eval` will bind that port and listen for connections. The environment is the state which defines the function bindings the clients can use. The server is expected to create an environment and define the appropriate symbols in that new environment to limit what functions clients can access.

B.2 Secure Evaluator Code

```scheme
;; The Secure Evaluator
;;
;; Created by:      Derek Atkins <warlord@MIT.EDU>
;;
;; $Source: /u3/warlord/src/media-bank/scm/RCS/secure-eval.scm,v $
;; $Author: warlord $
;;
;; Exported Function List:
;;
;; (secure-eval-add-connect-hook f)
;; (secure-eval-add-disconnect-hook f)
;; (secure-eval-add-init-hook f)
;;
;; (secure-eval-current-client)
;; (secure-eval-set-client-security token private-key)
;;
;; (secure-eval-loop port environment)
;;

(require 'dsys 'file 'crypto)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Global Secure-eval Definitions

(define secure-eval--connect-hooks '())
(define secure-eval--disconnect-hooks '())
(define secure-eval--init-hooks '())
(define secure-eval--pollblock (dsys-pollblock-create))
(define secure-eval--this-client '())
(define secure-eval--client-map (map-create))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Set function hooks

(define secure-eval-add-connect-hook
  (lambda (f)
    (set! secure-eval--connect-hooks
          (reverse (cons f (reverse secure-eval--connect-hooks))))))

(define secure-eval-add-disconnect-hook
  (lambda (f)
    (set! secure-eval--disconnect-hooks
          (reverse (cons f (reverse secure-eval--disconnect-hooks))))))

(define secure-eval-add-init-hook
  (lambda (f)
    (set! secure-eval--init-hooks
          (reverse (cons f (reverse secure-eval--init-hooks))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Obtain information about data structures
```

```
(define secure-eval-current-client
  (lambda ()
    secure-eval--this-client))

(define secure-eval-client-info
  (lambda ()
    (map-value secure-eval--client-map (stringout secure-eval--this-client))))

(define secure-eval-client-security
  (lambda ()
    (vector-lookup (secure-eval-client-info) 'security)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Security Parameters and Functions

(define secure-eval-set-client-security
  (lambda (token private-key)
    (let ((client (secure-eval-client-info)))
      (set-cdr! (vector-lookup-binding client 'security)
                (cond ((null? token)
                       '())
                      (#t
                       (let* ((info (read-key-exchange private-key token))
                              (block (block-create (car info) (cdr info))))
                         (vector
                          (cons 'pubkey private-key)
                          (cons 'privkey private-key)
                          (cons 'block block)))))))))

;; open and verify the request, returning the value or an error
(define read-request
  (lambda (request)
    (let ((crypto (secure-eval-client-security)))
      (packet->dtype (unseal-packet request crypto)))))

;; seal a reply to the client
(define seal-reply
  (lambda (reply)
    (let ((crypto (secure-eval-client-security)))
      (seal-packet (dtype->packet reply) crypto))))

;; perform a secure request, replying securely (exported function)
(define secure-eval
  (lambda (request environment)
    (let* ((req (exception-handler (read-request request)))
           (response (exception-handler (eval req environment))))
      (seal-reply response))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Process Client Data

(define server-login-client
  (lambda (server client)
```

```
    (map-insert! secure-eval--client-map (stringout client)
                 (vector (cons 'security '())))
    (for-each (lambda (f) (f server client)) secure-eval--connect-hooks)))

(define server-logout-client
  (lambda (server client)
    (map-delete! secure-eval--client-map (stringout client))
    (for-each (lambda (f) (f server client)) secure-eval--disconnect-hooks)))

(define server-process-packet
  (lambda (client packet env)
    (let ((security (secure-eval-client-security)))
      (if (null? security)
          (exception-handler (eval packet env))
          (secure-eval packet env)))))

(define server-process-clients
  (lambda (server env)

    (dsys-pollblock-clear! secure-eval--pollblock)
    (dsys-station-add-read! server secure-eval--pollblock)
    (dsys-pollblock-block secure-eval--pollblock)

    (define r (dsys-server-poll server))

    (define client (car r))
    (set! secure-eval--this-client client)

    (cond ((equal? (cdr r) 'login)
           (server-login-client server client))
          ((equal? (cdr r) 'pending)
           (define packet (exception-handler
                           (dsys-connection-read
                            (dsys-server-client server client))))
           (define ret (exception-handler
                        (server-process-packet client packet env)))
           (dsys-connection-write (dsys-server-client server client) ret))
          ((equal? (cdr r) 'logout)
           (server-logout-client server client))
          (exception (cons "unknown response from server" r)))))

(define secure-eval-loop
  (lambda (port env)
    (define server (dsys-server-create port))
    (for-each (lambda (f) (f server)) secure-eval--init-hooks)
    (while #t (printing-exception-handler
               (server-process-clients server env)))))
```

Appendix C

Transport Layer Code

C.1 Transport Functions

The following paragraphs define the functions available to a client of the transport server.

`mbank-open (name security)`: This function is used to open an object in the media bank. The name is the actual object handle on the server, which is obtained from the directory server. The transport server looks up the name and finds the local file(s) associated with the object, and then checks the security object for validity.

If the client is allowed to open this object, as specified in the security object, then the transport server will open the object and return a handle to the client, which the client can use to access this object.

`mbank-close (handle)`: This function is used to close an object. The server will close the resources associated with the object, and close the security ticket associated with this connection.

`mbank-bytes-read (handle size)`: This function is used to read data out of an object. It will read size bytes from the object, from the current position, and return it to the client in the form of a packet.

`mbank-bytes-seek (handle position)`: This function is used to seek to some position within an object file. The server will seek to the passed-in position in the object referred to by the handle. Subsequent reads from this object will happen from this location.

C.2 Object Server Code

```scheme
;; Media Bank Transport Server
;;
;; Created by:     Derek Atkins <warlord@MIT.EDU>
;;
;; $Source: /u3/warlord/src/media-bank/scm/RCS/transport.scm,v $
;; $Author: warlord $
;;

(require 'scheme 'unix 'gdbm 'map 'set 'file 'crypto 'secure-eval)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Global Server Definitions

;; setup a random seed for the random uids
(unix-srandom (unix-time))

(define client-map (map-create))          ; map of client->client env
(define senv (environment-create))        ; restricted server environment
(define server-port 20445)                ; the port on which this server listens

;; Setup the item server information
(define server-hostname (vector-ref (string-split (unix-hostname) ".") 0))
;;(define data-directory "/system/media-bank/server/data")
(define data-directory "/usr/tmp")
(define gdbm-filename (string-append data-directory "/"
                                     server-hostname ".gdbm"))
(define gdbm-file (gdbm-open gdbm-filename 'wrcreate))
(define dtype-name "descriptor.dtype")
(define partition-filename "/tmp/partitions")
(define serial-number 0)

;; The movie ticket server
(define mts-server-host "alphaville.media.mit.edu")
(define mts-server-port 41923)
(define mts-server-connection #f)
(define mts-crypto #f)
(define mts-cipher 'idea)
(define mts-cipher-key #f)
(define mts-pubkey #f)

;; Read in the server private key
(define keyfile
  "/mas/garden/grad/warlord/src/crypto-class/src/test/rsakey.dat")
(define keystr (iostream-open keyfile "r"))
(define private-key (privkey-create-from-iostream 'rsa keystr))
(iostream-close keystr)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Serial Numbers: set, update, current

(define serial-clear
```

```
  (lambda () (set! serial-number 0) #f))

(define serial-update
  (lambda () (set! serial-number (+ 1 serial-number)) #f))

(define serial-current
  (lambda () serial-number))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Database Manipulations

(define item-store
  (lambda (key item)
    (gdbm-store gdbm-file key item)
    (serial-update)))

(define item-delete
  (lambda (key)
    (gdbm-delete gdbm-file key)
    (serial-update)))

(define database-clear
  (lambda ()
    (serial-update)
    (for-each item-delete (item-list))))

(define database-add-file
  (lambda (filename)
    (define process-entry
      (lambda (entry)
        (gdbm-store gdbm-file (vector-lookup entry 'uniqueid) entry)))
    (define file (iostream-open filename "r"))
    (while (not (error? (process-entry (iostream-parse-dtype file)))) #f)
    (serial-update)))

;; Given a filename containing a list of filenames, load each file into
;; the database
(define database-load-files
  (lambda (filename)
    (database-clear)
    (define file (iostream-open filename "r"))
    (for-each database-add-file
              (string-split (iostream-read-string file) "\n"))
    (iostream-close file)))

(define concat-filename-list
  (lambda (files)
    (let ((names ""))
      (for-each (lambda (name) (set! names
                                     (string-append names " " name)))
                files)
      names)))

;; Take the list of partitions in the partitions file, find all the
```

```
;; dtypes (look for the dtype-name file), and put all of them into
;; the output file.
(define find-items
  (lambda (partitions outfile)
    (define file (exception-handler (iostream-open partitions "r")))
    (define file-contents (cond ((error? file)
                                 "/")
                                (#t
                                 (iostream-read-string file))))
    (if (not (error? file))
        (iostream-close file))
    (unix-system (string-append
                  "find "
                  (concat-filename-list
                   (string-split file-contents "\n"))
                  " -xdev -name " dtype-name " -print > " outfile))))

;; Load the database with all the items found in the partitions in the
;; filename given to this function
(define load-items
  (lambda (filename)
    (define tempfile (string-append "/usr/tmp/" server-hostname ".dtype-list"))
    (unix-system (string-append "rm -f " tempfile))
    (find-items filename tempfile)
    (database-load-files tempfile)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Item Server Functions

(define item-fetch
  (lambda (key)
    (gdbm-fetch gdbm-file key)))

(define item-list
  (lambda ()
    (gdbm-listkeys gdbm-file)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Object information -- given a name find object info and things out it

;; Get the actual item information given its name.  This should go to
;; the item server on this machine and ask for the item by name, and
;; return what it gets.
;; XXX -- for now just return the name!
(define get-mbank-object
  (lambda (name)
    (item-fetch name)))

;; Return a file path from a url format file://localhost/...
(define path-from-url
  (lambda (url)
    (let* ((url-vec (string-split url ":"))
           (type (vector-ref url-vec 0))
           (path (vector-ref url-vec 1)))
```

```
    (if (not (string=? type "file"))
        (exception "unknown URL type " type))
    (if (string-contains? (string-subrange path 0 2) "//")
        (let ((path-vec (string-split path "/"))
              (path ""))
          (vector-remove! path-vec 0)
          (for-each (lambda (p)
                      (set! path
                        (string-append path "/" p)))
                    (vector->list path-vec))
          path)
        path))))

;; Return the path for an object's data.  Right now, assume that the
;; object is a valid path, but this can (and will) change to pass
;; in a media-bank item-server object, and figure out the local
;; file system pathname
(define data-path-from-object
  (lambda (item)
    (let ((data (exception-handler (vector-lookup item 'data))))
      (if (error? data)
        (exception "Cannot find data file")
        (path-from-url data)))))

;; Return the path for an object's index file, assuming it has such
;; a beast.  If it exists, then this would allow a client to read
;; data in time, rather than in space, from this object.
(define index-path-from-object
  (lambda (item)
    (let ((index (exception-handler (vector-lookup item 'index))))
      (if (error? index)
        "/*bad-path*"
        (path-from-url index)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Client information -- create, access, and destroy client contexts

;; A new client has connected.  Generate state for that client

(secure-eval-add-connect-hook
  (lambda (server client)
    (map-insert! client-map (stringout client)
                 (vector
                  (cons 'token-map (map-create))))))

;; A client has disconnected.  Let's clean up!
(secure-eval-add-disconnect-hook
  (lambda (server client)
    (let* ((token-map (vector-lookup (map-value client-map (stringout client))
                                     'token-map))
           (uids (map-listkeys token-map)))
      (for-each (lambda (uid)
                  (let ((value (map-value token-map uid)))
                    (do-close value)
```

```
                        (map-delete! token-map uid)))
                    uids)
     (map-delete! client-map (stringout client)))))

;; Return the client environment information for the current client
(define client-info
  (lambda ()
    (map-value client-map (stringout (secure-eval-current-client)))))

;; Return the client token-map information for the current client
(define client-token-map
  (lambda ()
    (vector-lookup (client-info) 'token-map)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Movie Ticket Validation

(define make-rpc
  (lambda (request)
    (if mts-server-connection
        (let ((resp (exception-handler (do-rpc request))))
          (if (error? resp)
              (attempt-connect-mts request)
              resp))
        (attempt-connect-mts request))))

(define do-rpc
  (lambda (request)
    (exception-handler
     (packet->dtype
      (unseal-packet
       (dsys-connection-rpc mts-server-connection (seal-packet
                                                   (dtype->packet request)
                                                   mts-crypto))
       mts-crypto)))))

(define attempt-connect-mts
  (lambda (request)
    (if (connect-mts)
        (do-rpc request)
        (exception "Cannot connect to movie ticket server"))))

(define set-mts-crypto
  (lambda ()
    (if mts-crypto
        #f
        (let ((key (make-random-key mts-cipher)))
          (set! mts-crypto (vector
                            (cons 'pubkey mts-pubkey)
                            (cons 'privkey private-key)
                            (cons 'block (block-create mts-cipher key))))
          (set! mts-cipher-key key)
          #t))))
```

```scheme
(define connect-mts
  (lambda ()
    (define conn (exception-handler
                  (dsys-connection-create mts-server-host mts-server-port)))
    (cond ((error? conn)
           (set! mts-server-connection #f)
           #f)
          (#t
           (set! mts-server-connection conn)
           (define pubkey (dsys-connection-rpc conn '(server-public-key)))
           (if (not mts-pubkey)
               (set! mts-pubkey
                     (pubkey-create 'rsa (car pubkey) (cdr pubkey))))
           (set-mts-crypto)
           (printing-exception-handler
             (dsys-connection-rpc conn
                                  (list 'set-security
                                        (list 'quote
                                              (setup-key-exchange
                                               mts-pubkey
                                               mts-cipher
                                               mts-cipher-key)))))
           #t))))

;; Check if an object is valid given a ticket
;; XXX: this needs to be fixed in a future version.
(define object-in-ticket?
  (lambda (object ticket)
    (define pag (vector-lookup (car ticket) 'pag))
    #t))
;;    (set-contains? pag (vector-lookup object 'uniqueid))))
;; If the MTS ever gets a full set of objects, then use this.
;; Otherwise, we could do a reverse-map, change the data in here to be
;; a string rather than a set, and then see if the object is a part
;; of the program defined in the ticket.

;; This function validates a security object.
;;
;; First, check if the object has a price.  If the price is 'free then
;; approve the download.  Next, check the ticket's validity.  Finally,
;; check to see if the ticket is valid for this object
(define validate-security
  (lambda (object security)
    (define price (exception-handler (vector-lookup object 'price)))
    (if (and (not (error? price))
             (equal? price 'free))
        #t
        (let ((val (printing-exception-handler
                    (make-rpc (list 'validate-ticket (list 'quote security)
                                    (vector-lookup object 'uniqueid))))))
          (if (error? val)
              #f
              (if val
                  (object-in-ticket? object security)
```

```
                       val))))))

;; This function tells the MTS that we're done with an object
(define object-done
  (lambda (object security)
    (make-rpc (list 'turnin-ticket (list 'quote security)
                        (vector-lookup object 'uniqueid)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Set Security Operations for this connection -- choose confidentiality
;;  and integrity methods for this connection

(define set-security
  (lambda (token)
    (secure-eval-set-client-security token private-key)))

(define server-public-key
  (lambda ()
    (pubkey-extract private-key)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Code to open and close -- these are the "exported" routines, plus
;; their helper functions

;; Generate a UID for a client
(define generate-uid
  (let ((uid-size 16777216))                    ;; 2^24
    (lambda ()
      (unix-random uid-size))))

;; Actually perform the open
(define do-open
  (lambda (item ticket)
    (let* ((info (vector (cons 'item item) (cons 'ticket ticket))))
      (set! info (vector-concat info (mbank-open-data item)))
      (set! info (vector-concat info (mbank-open-index item)))
      info)))

;; Actually perform the close
(define do-close
  (lambda (value)
    (object-done (vector-lookup value 'item) (vector-lookup value 'ticket))
    (mbank-close-data value)
    (mbank-close-index value)))

;; The global open -- this will then do the appropriate opens
;; for the valid functions for this type of object
(define mbank-open
  (lambda (name security)
    (let ((uid (generate-uid))
              (token-map (client-token-map))
              (item (get-mbank-object name)))
      (cond ((validate-security item security)
                (map-insert! token-map uid (do-open item security))
```

```
                uid)
               (#t
                (exception "Authorization failed"))))))

;; The global close -- this will do the appropriate closes for this
;; uid (object) and remove it from the token-maps.
(define mbank-close
  (lambda (uid)
    (let* ((token-map (client-token-map))
           (value (map-value token-map uid)))
      (do-close value)
      (map-delete! token-map uid))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Open the stream for an object

;; (mbank-open-data object) -- returns a vector of data cons cells
(define mbank-open-data
  (lambda (item)
    (let* ((path (data-path-from-object item))
           (f (exception-handler (iostream-open path "r"))))
      (if (error? f)
          (exception (stringout "Error opening file " path)))
      (vector
       (cons 'data-stream f)))))

;; (mbank-open-index object) -- returns a vector of index cons cells
(define mbank-open-index
  (lambda (item)
    (let* ((path (index-path-from-object item))
           (f (exception-handler (iostream-open path "r"))))
      (vector
       (cons 'index-stream
             (if (error? f)
                 '()
                 f))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Close the streams, given an object vector

;; (mbank-close-data value)
(define mbank-close-data
  (lambda (value)
    (let ((f (vector-lookup value 'data-stream)))
      (if (not (null? f))
          (iostream-close f)))))

;; (mbank-close-index value)
(define mbank-close-index
  (lambda (value)
    (let ((f (vector-lookup value 'index-stream)))
      (if (not (null? f))
          (iostream-close f)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Get the proper stream from the uid for a client request

;; Get the data stream from a uid.
(define get-data-stream
  (lambda (uid)
    (let* ((token-map (client-token-map))
               (value (map-value token-map uid))
               (item (vector-lookup value 'item)))
      (vector-lookup value 'data-stream))))

;; Get the index stream from a uid.
(define get-index-stream
  (lambda (uid)
    (let* ((token-map (client-token-map))
               (value (map-value token-map uid))
               (item (vector-lookup value 'item)))
      (vector-lookup value 'index-stream))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BYTE functions:  Read and seek by byte offset

;; (mbank-bytes-read uid length) -- returns a packet
(define mbank-bytes-read
  (lambda (uid length)
    (let ((f (get-data-stream uid)))
      (if (not (null? f))
            (iostream-read f length)
            (printout "No data stream for handle " uid "\n")))))

;; (mbank-bytes-seek uid position)
(define mbank-bytes-seek
  (lambda (uid pos)
    (let ((f (get-data-stream uid)))
      (if (not (null? f))
            (iostream-seek f pos)
            (printout "No data stream for handle " uid "\n")))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TIME functions:  Read by time offset.  Not all items have this
;;  ability.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Setup the environment

(define add-symbols
  (lambda (env symbols)
    (for-each (lambda (s) (apply environment-define (list env s s))) symbols)))

(add-symbols senv
                '(mbank-open mbank-close mbank-bytes-read
                              mbank-bytes-seek set-security item-list
                              item-fetch serial-current server-public-key
```

```
                    quote cons vector list printout stringout
                    current-environment and or not))

(load-items partition-filename)

(secure-eval-loop server-port senv)
```

Bibliography

[1] Stefan Brands. An Efficient Off-line Electronic Cash System Based on the Representation Problem.

[2] D. Atkins, M. Graff, A. Lenstra, P. Leyland. The Magic Words Are Squeamish Ossifrage. In *AsiaCrypt*, 1994.

[3] J. Schiller D. Eastlake III, S. Crocker. Randomness Recommendations for Security. RFC 1750, December 1994.

[4] Klee Dienes. The Dtype++ Reference Manual. Internal document, 1995.

[5] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[6] Federal Register. *Proposed Federal Information Processing Standard For Digital Signature Standard (DSS)*, 30 August 1991. v. 56, n. 169.

[7] Federal Register. *Proposed Federal Information Processing Standard For Secure Hash Standard*, 31 January 1992. v. 57, n. 21.

[8] Niels Ferguson. Single Term Off-Line Coins.

[9] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February, 1988.

[10] X. Lai. On the Design and Security of Block Ciphers. *ETH Series in Information Processing*, 1, 1992. Konstanz: Hartung-Gorre Verlag.

[11] Andrew Lippman. The Media Bank. Internal memo.

[12] Gennady Medvinsky and B. Clifford Neuman. NetCash: A design for practial electronic currency on the Internet.

[13] National Bureau of Standards, Washington, D.C. *Data Encryption Standard*, Federal Information Processing Standards Publication 46 edition, 1977. Government Printing Office.

[14] L. Adleman R. Rivest, A. Shamir. A method for obtaining digital signatures and public key cryptosystems. *CACM*, 21(2):120–128, February 1978.

[15] R. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992. MIT Laboratory for Computer Science.

[16] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Technical report, M.I.T. Project Athena, "1985, 1986, 1987".