Rochester Institute of Technology
RIT Scholar Works

Theses Thesis/Dissertation Collections

2006

Security in an ad hoc network using many-to-many invocation

Jefferson Tuttle

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Tuttle, Jefferson, "Security in an ad hoc network using many-to-many invocation" (2006). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Department of Computer Science

Rochester Institute of Technology

MS Project

Security in an Ad Hoc Network using Many-to-Many Invocation

Revision: 1.2

Date: May 9, 2003

Author: Jefferson S. Tuttle (jst1734@cs.rit.edu)

Approvals:

____

Chair Alan R. Kaminsky Date

____

Reader Hans-Peter Bischof Date

____

Observer Stanislaw Radziszowski Date

Abstract

There is an increasing need to secure electronic communication. This need is being met on workstations and servers, which have large amounts of memory, and a dedicated network connection. Securing communication in mobile devices is a relatively new research area, and has different requirements than workstations and servers. This project attempts to address these new requirements by incorporating security into the Many-to-Many Invocation (M2MI) library developed at the Rochester Institute of Technology (RIT) [KB02].

Specifically, this project addresses adding encryption and decryption of M2MI method invocations using session keys. It also addresses the methods of establishing and maintaining session keys based on communicating members sharing a common group authentication key. This project leaves the establishment and maintenance of the common group authentication key(s) as a separate step, outside the scope of this project.

Table of Contents

1 Introduction

This project focuses on two steps for securing communication. The first step is authenticating to whom you are talking. The second is hiding the data that is being communicated so that only the sender and receiver(s) know what is being communicated. These ideas are nothing new. Authentication and encryption have been around for a while, and there are some strong algorithms in place for performing these two steps. A relatively new area of security is secure group communication. Within this field, secure group communication in an ad hoc network is an area of current research. Ad hoc networks bring two challenges to secure group communication. The first is the lack of a central server. Central servers are typically used to provide the authentication step in establishing a secure communication channel. Maintaining membership lists and storing public keys of group members on the devices that want to communicate can overcome this problem. This has drawbacks, especially in mobile ad hoc networks. In mobile ad hoc networks, the memory sizes of communicating devices are typically quite limited, and are not capable of storing large membership lists or a large number of keys. These limited capabilities are the second challenge.

This project overcomes these challenges by splitting the authentication step into two phases. The first phase involves establishing and maintaining group authentication keys. Each device that will communicate in a mobile ad hoc network will store a group authentication key (or key pair) on the device for each group that it belongs to. This group authentication key will be established and maintained while the device can communicate with a central server (either directly or via a host computer, then downloaded from the host computer using a secure communication channel, e.g. PDA cradle, IrDA, etc.). As members of each group are added or removed, the device will get updates to the group authentication key when it connects to the central server (or host computer). These group authentication keys must be kept securely on the device to prevent compromise. Another project at RIT is currently addressing these issues [Bi03].

The second phase involves using these group authentication keys to establish a session key that will be used during a single group communication session. All members of a group that have received the group authentication key will be able to participate. Each member will use the group authentication key to perform an encrypted key exchange in establishing a session key. Only members who have the authentication key will be able to comprehend the session key. The protocol for performing this encrypted key exchange is described in this paper.

The session key that is established with the encrypted key exchange will then be used to secure all communication among the group. For this particular project, the data that is encrypted with the session key is the Many-to-Many invocations of the various test clients.

2 M2MI Overview

This project focuses on M2MI, and as such, some background is needed on how M2MI works in order to understand the changes that were made. This section gives a brief description of M2MI. A more complete description is available in the OOPSLA paper on M2MI [KB02].

The M2MI library is broken up into two layers. The top layer is called the M2MI layer, and is what the applications use to interface with the library. The lower layer is called the M2MP layer, and is used for routing M2MI messages between applications.

M2MI supports three types of communication: one-one, one-many, and many-many. As with any electronic communication, an endpoint needs to be created in order to send out messages. For M2MI, there are three types of endpoints: unihandles, multihandles, and omnihandles. Unihandles are used for point-to-point communication. Multihandles are used to multicast messages to a particular group of users. Omnihandles are used to broadcast messages to everyone listening.

As was implied with omnihandles, there must be an application out there listening for these messages being sent. In M2MI, an application listens for messages by exporting an object that implements a specific interface. The M2MI library maintains an export map, which maps the interface name to the object, or set of objects, that implement that interface within a particular application. This way, when a message is received that contains that interface name, it can be forwarded to the object, or set of objects, that exported that interface. In addition to storing the mapping of the interface name to the object, the M2MI library also creates a message filter object, which contains a hash of the interface name, and a "magic number" which is used to identify the message as an M2MI message. This message filter object provides a filter in the M2MP layer of which messages the application would like to receive. Only messages that contain exported interfaces will be sent from the M2MP layer to the M2MI layer. The M2MI layer is then responsible for calling the appropriate method on the appropriate objects that have exported the interface contained in the received message.

The following example helps illustrate how this works for omnihandles. First, application A exports interface I with object O. Assume interface I has method M. This creates a mapping in the M2MI layer between interface I and object O. Then, the M2MI layer creates a filter in the M2MP layer for interface I. Application B then creates an omnihandle for interface I, and invokes method M on the omnihandle. The M2MI layer then creates a message containing the M2MI magic number, a hash code of the interface name associated with the omnihandle, and method M's arguments. The M2MP layer then broadcasts the message over the channel associated with the M2MP layer (this association is done at initialization time). The M2MP layer in application A receives the broadcast message, compares the magic number and interface name hash code from the message with those in its message filter list. It finds a match, and sends the message up to the M2MI layer. The M2MI layer then looks up the interface name in its export map, and finds object O. The M2MI layer then invokes method M on object O.

The above gives a basic example of how M2MI method invocation works with omnihandles. Multihandles and unihandles work a little differently. With multihandles and unihandles, the sender and receivers need to know about each other. For unihandles, one application will create a unihandle, associate it with an object that implements a particular interface, and send the unihandle to other applications. The unihandle can be sent using another unihandle, a multihandle, or an omnihandle. When the receiver receives the unihandle, it then has a handle to a single object that implements a particular interface. When a method is invoked on that unihandle, a message is sent back to the application that associated an object to the handle, and the method invoked on the unihandle is then invoked on the object associated with the unihandle in the original application. The following example describes this in more detail.

Application A creates a unihandle U for interface I, and associates it with object O (all done in one method call to the M2MI library). Assume interface I contains method M. Unihandles associate unique Exported Object IDs (EOIDs) with objects, instead of interface names. The EOID for unihandle U and the object O are added to the export map in application A's M2MI layer, and a message filter is added to the M2MP layer that contains the EOID for unihandle U and the M2MI magic number. Application A then creates an omnihandle to another interface J. Application A then invokes a method N on the omnihandle, passing unihandle U as an argument. The unihandle is then sent to all applications that exported interface J. Application B is one of those applications that exported interface J. Application B receives unihandle U, as described in the above omnihandle example. Application B then invokes method M on unihandle U. This causes the M2MI layer in application B to create a message that contains the M2MI magic number, the EOID of unihandle U, and the arguments to method M. The M2MP layer then broadcasts the message over the channel associated with the M2MP layer (this association is done at initialization time). When application A receives this message, the M2MP layer matches the magic number and EOID with one of its message filters, and sends the message up to the M2MI layer. The M2MI layer then finds the EOID in its export map, which maps to object O. The M2MI layer then invokes method M on object O.

Multihandles work very similar to unihandles. Like unihandles, multihandles have a unique EOID. Multihandles must also be created in one application and sent to other applications (or to other objects within the same application). However, unlike unihandles which can only be associated with a single object, multihandles can be associated with multiple objects. A multihandle can be created in application A and associated with object X, send to application B and associated with object Y, sent to application C and associated with object Z, and so on. Any of the applications that receive the multihandle can invoke a method on that handle, say method M. A message is created with the EOID of the multihandle, and broadcast on the channel associated with the M2MP layer. All applications that received the multihandle and associated objects with the multihandle, will have method M invoked on their object, i.e. X.M, Y.M, and Z.M will all be invoked.

3 Issues Addressed in this Project

The following issues were addressed in order to establish a secure connection and achieve the basic security requirements of identification, data confidentiality, and data integrity. Each issue is described in later subsections, as referenced by each issue.

- Establish a group authentication key. (Group Key Establishment)
- Maintain the group authentication key. (Group Key Management)
- Establish a communication session key. (Session Key Establishment)
- Encrypt/decrypt M2MI method invocations. (M2MI Integration)

The above basic issues can be broken down into more detailed issues. These detailed issues are described in subsections referenced by each detailed issue.

- How to handle multiple devices starting a communication session at the same time. (Multiple Startup)
- How to merge communication sessions that use the same group authentication key, but were formed

separately from each other. (Group Merge)

- Rolling over communication session keys so that the same session key is not used for long periods of time. (Key Rollover)
- Defining/choosing an encryption/decryption algorithm that can be implemented on small devices. (Encryption Algorithm)
- Differentiating between M2MI messages encrypted with the session key and random messages sent by an impostor. (Impostor Detection)
- Measuring memory and CPU usage to determine if the algorithms developed are appropriate for small devices. (Measurements)

Finally, in order to show that this project works, test clients were developed in order to show the full integration of the secure channel. Many of these test clients were derived from existing M2MI test clients. (Test Client)

3.1 Group Key Establishment

A significant amount of research was performed on this project to determine how to establish a group key. Several techniques were reviewed, including some that are being developed at RIT. Due to the variety of techniques available, and the current research at RIT related to this topic, this issue was left out of the scope of this project. Some of the findings are discussed in this section and the next section. However, for purposes of limiting the scope of this project, and not interfering with other research at RIT, it is assumed that the group authentication key will be established by other means outside of this project, and that the authentication key will be available on the device for the software developed on this project to access.

The rest of this section gives a brief discussion of some of the methods that could be used to establish a group authentication key. This discussion is broken down into two sections. The first discusses the traditional approach of using a central server for maintaining the group keys. The second discusses the establishment of a group key among a group of communicating devices, where each node plays a part in defining the group key.

3.1.1 Central Server Approach

This approach assumes that a central server maintains the group authentication key, which is downloaded to devices that want to perform secure M2MI. The method of downloading the key may vary between devices, as described in this section.

On some devices, a secure connection to a central server may be possible directly from the device (e.g. Public Key Infrastructure). In this case, the device can establish the secure connection and download the group authentication key directly from the central server.

On some devices, a secure connection to a central server may be impossible to establish directly from the device. However, if the device has a direct connection to a PC host computer, e.g. cradle, IrDA, etc., the host computer can download the group authentication key from the central server, then download the group authentication key to the device.

3.1.2 Group Communication Approach

There are many research projects that have defined methods for members of a communication group to establish a group key by each member of the group providing a piece of the key. Some of these include one-way accumulators [BdM93], Group Diffie-Hellman [STW96], key agreement for dynamic peer groups [STW00], tree based Group Diffie-Hellman [KPT00], and algorithms based on zero knowledge proof of identity [FFS87]. Many of these approaches are depend on a reliable communication system, or for the authentication to be performed outside of the protocol they define. These requirements were not acceptable for this project.

3.2 Group Key Management

The group authentication key(s) for a device is maintained in non-volatile memory on the device. For purposes of this project, it is assumed that these keys will be stored in a disk file. However, the interface for retrieving the group authentication key from non-volatile memory does not assume that it is a disk file. A class has been written that retrieves the authentication key. This class currently retrieves it from a disk file, but this can be easily modified to retrieve it from some other storage facility.

If members are added to or removed from the group, a new group authentication key should be established, and updated on the device. This allows for forward and backward secrecy. This step is outside the scope of this project.

Any changes to a group authentication key will be propagated to the device using the same methods described in Section 3.1 Group Key Establishment.

3.3 Session Key Establishment

In order to provide the most flexibility to an application, the session key that will be used to encrypt and decrypt method invocations is maintained by each handle, and for each exported interface. Whenever an application creates a handle or exports an interface, a session key for that handle/interface will be established. This is handled automatically be the M2MI library, and a mapping is maintained in the library between a secure group name and the session key associated with that name, so that only one session key is generated per secure communication group.

This section describes the protocol of establishing a session key. This protocol takes place "under the hood" when a secure handle is created, or when an interface is exported using the secure methods of the M2MI library. The new security methods of the M2MI library are discussed in Section 5 Detailed M2MI Changes.

The following interface is used to establish a session key.

```
interface SessionKeyEstablisher {

      void requestSessionKey ( String groupName );

      void reportSessionKey ( String groupName, byte[] encryptedKey,

long keyID);

}
```

The secure M2MI library will use interface `SessionKeyEstablisher` to obtain a session key. When a session key is created, the library will invoke method `requestSessionKey` on an omnihandle for interface `SessionKeyEstablisher`, so that all devices out there will process the invocation and potentially send back the session key. A timer will then be set to limit the amount of time the device will wait to obtain a session key. The library will also export an object that implements interface `SessionKeyEstablisher`. If method `reportSessionKey` gets invoked on the exported object before the timer goes off, the timer will be canceled, and the session key received in `reportSessionKey` will be used as the session key. If the timer goes off, the device will generate its own session key, and invoke `reportSessionKey` to tell any devices that might be out there that they have established a new session key.

Each device will invoke `reportSessionKey` under the following conditions:

- Upon reception of a `requestSessionKey` invocation, to let other devices know what the current session key is. However, to avoid a broadcast storm [NTCS99], each device will create a short, random back-off timer. When that timer expires, it will respond with a `reportSessionKey` invocation. If the device receives a `reportSessionKey` invocation before the back-off timer expires, the timer will be canceled, and it will not invoke `reportSessionKey`.

- Periodically, to help with merging of communication groups and to let other devices know that the key is still active.

- After a given expiration period, at which point in time the device will generate a new session key and broadcast the new key. Whenever a device receives a `reportSessionKey` with a higher key id, it will use the new session key and key id received, and reset its expiration timer. This expiration timer is first set on the first `reportSessionKey` the device receives, or when the `requestSessionKey` timer expires. It is reset every time a new key is received, or when it generates a new session key.

More details on these and other conditions are provided in Section 3.5 Protocol Issues.

3.4 M2MI Integration

The existing M2MI library had to be modified extensively in order to continue to provide clients with the greatest flexibility, and maintain the same APIs that were already published by the M2MI library. The details of these changes are provided in Section 5 Detailed M2MI Changes.

3.5 Protocol Issues

This section describes issues that need to be resolved during the project associated with the protocol used to establish a session key, as well as issues related to data confidentiality and data integrity. The first paragraph of each of the following subsections describes the issue that needs to be resolved. The remaining paragraphs in each subsection describe how these issues were resolved.

3.5.1 Multiple Startup

ISSUE: If many clients startup at about the same time, how will they arrive at a single sessionKey?

As described in Section 3.3 Session Key Establishment, each device will invoke `requestSessionKey` on an omnihandle for interface `SessionKeyEstablisher`, so that all devices out there will process the invocation and potentially send back the session key. Each device will then listen for a `reportSessionKey` invocation. If a `reportSessionKey` invocation occurs within a given timeout period, the key received will be used as the current session key. If no key is received, the device will create its own key with a unique key id, and broadcast the new key and key id using `reportSessionKey`. If another device starts up about the same time, there are three possible outcomes:

- Second device receives `reportSessionKey` of first device within its timeout period. In this case, both devices will use the session key and key id sent by the first device.

- Second device starts up before the first device, and declares its session key and key id before the first device times out waiting for `reportSessionKey`. This is the same as case 1, but looking at it from the other device's perspective.

- Both devices timeout before receiving any `reportSessionKey` invocations. In this case, both devices will invoke `reportSessionKey`. In this case, the `reportSessionKey` invocation containing the largest key id wins. The device receiving the larger key id will store the incoming key and key id.

This scenario was given with two devices. This could easily be extended to N devices by looking at any two devices in the set of N, and deciding the outcome of those two devices.

3.5.2 Group Merge

ISSUE: If a session key has been established, and a new session key is being reported by `reportSessionKey`, how do these different "groups" get merged?

- Nothing special needs to be done in this case. If a device receives a `reportSessionKey` invocation with a larger key id than is currently associated with the current session key, the new session key becomes the current session key, and new key id is saved. In other words, devices in the group that is using the smaller key id will replace their key and key ids with those of the key and key id associated with the larger key id.

3.5.3 Key Rollover

ISSUE: In cryptography, the longer a key is used, the more likely it is that the key may be cracked. The security package developed in this project must take care to limit the amount of time that a session key is used. If a session key has been used for more than a fixed amount of time (which may be configurable), then it should invalidate its key and create a new session key. This value is currently a hard coded constant. This could easily be changed to be retrieved from a system property.

- This is easily accomplished with this protocol. When a device determines that its session key has been used for longer than its [configurable] amount of time (i.e. its expiration timer expires), it will generate a new session key and a new, higher, key id. The new key and key id are then broadcast using `reportSessionKey`. All other devices in the group will receive the new key id, see that it is larger, and switch over to using the new session key and key id. The timeout used by each device will be based on a fixed timeout, but will have some randomness to try to reduce the probability that more than one device has its key expire at the same time. If this isn't done, it would be very likely that all devices expire their keys at about the same time, and they would all try to generate a new key, creating unnecessary network traffic.

3.5.4 Encryption Algorithm

ISSUE: Need to define or choose an encryption/decryption algorithm that can be implemented on small devices. This algorithm should use fairly large keys for the group authentication key, and relatively small (64-256 bits) for the session key. Perhaps the key sizes can be configurable, within these defined limits. The algorithm should be able to run on small devices, which means that it should be able to run in a reasonable amount of time on a slow CPU, and should not require a lot of memory to process. The processing speed and memory requirement limits need to be defined.

The Java 2 Standard Edition (J2SE) platform offers many standard encryption algorithms that can be used with the session key. The Java 2 Micro Edition (J2ME) currently doesn't support the cryptographic extensions available in the J2SE platform. However, additional profiles are continuously being added to the J2ME platform. One of these will eventually support the cryptographic needs of this project.

For this project, the `java.security` and `javax.crypto` packages were used for implementing the encryption, decryption, and secure hash algorithms. This project uses the SHA-1 secure hash algorithm for generating message digests. This project uses DES for the encryption and decryption algorithm used with the session key. These algorithms are stored as public constant strings in the `SessionKey` class of the edu.rit.m2mi.security package, and can easily be changed. This project uses Triple-DES for the encryption and decryption algorithm used with the authentication key. These algorithms are stored as public constant strings in the `AuthenticationKey` class of the edu.rit.m2mi.security package, and can easily be changed.

3.5.5 Impostor Detection

The security package should be able to differentiate between M2MI messages encrypted with the session key and random messages sent by an impostor.

To overcome this issue, each message that needs to be encrypted will be sent through a one-way hash to create a fixed length message digest. This message digest will be concatenated to the original message. The combined message and message digest will then be encrypted with the appropriate key (session key or group authentication key, depending on which type of message is being encrypted). On the receiving end, the message will be decrypted with the appropriate key. The message will be separated into the original message and the fixed length message digest. The original message will be sent through the one-way hash function, and compared with the message digest. If they match, the message is authenticated, and can be used. If they don't match, the message is discarded.

3.5.6 Data Integrity

Messages sent with secure M2MI should be guaranteed to be delivered without modifications being made by an intruder.

This issue is overcome by using the one-way hash described in Section 3.5.5 Impostor Detection. By adding a message digest (the output of the one-way hash) to the message before encrypting the message, the receiver can be assured that the data is intact if the message decrypts successfully, and the message digest received matches the message digest that is computed from the received message.

4 Overview of M2MI Changes

One of the goals of this project was to preserve the current APIs for M2MI intact, and to add APIs for performing secure M2MI. Another goal of this project was to be able to use standard M2MI without having to import any of the security features that were added in this project. This was accomplished by leaving all of the APIs in the standard M2MI alone, and to add a new object that defined the APIs for secure M2MI. More details on these changes are covered later in this section.

A third goal of this project was to leave the M2MP layer alone, and to only modify the M2MI layer. This goal was achieved by defining a new message format that conformed to the requirements of the M2MP layer. The M2MP layer requires the M2MI layer to define and register a MessageFilter object. This message filter object encapsulates a message prefix. When a message is received by the M2MP layer, the M2MP layer looks through all of its MessageFilter objects to find one that has a matching message prefix. If one is found, the message is passed up to the M2MI layer.

For secure M2MI, the contents of the message prefix were changed to differentiate secure M2MI messages from standard M2MI messages. This allowed the M2MP layer to remain unchanged, yet allow the M2MP layer to forward the messages to the secure M2MI code rather than the original M2MI code.

The format of the message prefix is very similar to the message prefix of the original M2MI code. The original message prefix contained a magic number and a hash code. The magic number was a 4 byte representation of the string "M2MI". The hash code was a 4 byte hash code of either the Exported Object ID, or a 4 byte hash code of the interface name, depending on how the handle was exported. These message prefixes are maintained, and are still used by the standard M2MI code. The secure M2MI message prefix also has a magic number and a hash code. The magic number for secure M2MI messages is a 4 byte representation of the string "M2MJ". The hash code is still 4 bytes, and is the sum on the same 4 byte hash code as standard M2MI and the 4 byte hash code of the string representing the secure group name used to establish the session key.

One major change to the M2MI library was the representation of a handle. In standard M2MI, a handle encapsulates an interface name and an EOID. When a message is sent using that handle, all objects that were exported using that same interface name or EOID will invoke the method defined in the message sent using that handle. In secure M2MI, the handle still encapsulates the interface name and EOID. However, the handle also encapsulates a secure group name. The secure group name is a string that represents the name of the security group that is communicating with the same session key, and whose members share a common authentication key. When methods are invoked on that handle, the messages are encrypted using a session key associated with that secure group name. In addition to serving to lookup the session key, the secure group name is placed into the messages that represent that method invocation, and, as mentioned in the previous paragraph, its hash code is part of the message prefix of the message containing the method invocation.

The recipient of the secure method invocation message will then use the hash code of the secure group name and the EOID or interface name to lookup a message filter associated with that hash code. Once found, the secure group name will be used to lookup the session key in the receiver so that the message can be decrypted and authenticated.

- While the representation of the handle has changed, and the base class for all handles has changed, the user will still interface with the unihandle, multihandle, and omnihandle classes. When a user creates one of these handles, the secure M2MI code will synthesize the appropriate handle class (uni-, multi-, or omni-handle), but instead of these handles being derived from a `BaseHandle` class, which does the encapsulation of the EOID and interface name, these handle classes will be derived from a SecureHandle class, which encapsulates the EOID, interface name, and secure group name. The SecureHandle also knows to use the new secure M2MI methods for sending method invocations so that the method invocations will be encrypted and decrypted.

The last major change that is exposed to the user is the API for exporting objects and creating handles. There are many methods available in standard M2MI for exporting objects and creating handles. All of these methods are still available in secure M2MI, but each method requires an additional parameter, the secure group name. For instance, when an object is exported, the user would pass the interface name that the object implements, the object itself, and the secure group name.

5 Detailed M2MI Changes

Where possible, the existing M2MI code was used as is. However, in order reduce the amount of duplicated coded, many of the classes were modified. Many of the classes were modified simply to change the protection on data members or methods from private to protected, so that classes could extend the standard M2MI functionality to add security features. In one case, the M2MI class, the guts of the class were moved to a base class, M2MIBase, to share the functionality between the standard M2MI and secure M2MI implementations.

For a complete description of the public interfaces to the objects modified or added to the M2MI library, please refer to the javadoc included in the updated M2MI library. This section describes the changes to the existing classes, and provides an overview of each of the classes that were added.

- The original M2MI class contained the interfaces for initializing the M2MI library, exporting objects, unexporting objects, creating handles, and various internal methods. This class maintained a reference to the underlying M2MP layer, using the Protocol object. This class also performed the synthesis of MethodInvoker classes, which perform the actual method invocations on objects. All interaction with the underlying M2MP layer and synthesis of MethodInvoker classes has been moved to a new M2MIBase class. The rest of the functionality is maintained in what is left of the original M2MI class, which now extends M2MIBase. This functionality is duplicated in a new SecureM2MI class, but with the addition of requiring a secure group name to interfaces for exporting and unexporting objects, and for creating handles. In addition, many of the internal methods also require a secure group name.

- The original M2MI library used an ExportMap to map an interface name or EOID to the object or set of objects that were exported. A new SecureExportMap was created to map the combined interface name/secure group name, or EOID/secure group name, to the object or set of objects the were exported for the secure group name. The original ExportMap was left mostly untouched.

- The original M2MI library used a HandleSynthesizer class to synthesize unihandles, multihandles, and omnihandles. These handles all extended from a base class called BaseHandle. A new SecureHandleSynthesizer class was added, which is extended from HandleSynthesizer. The HandleSynthesizer class contains the same methods as before, and many of the methods are reused by the SecureHandleSynthesizer class. Some of the methods in HandleSynthesizer has to be modified to allow for more commonality. The main functionality added by SecureHandleSynthesizer was for synthesizing the SecureHandle base class. To reiterate, all secure M2MI handles, i.e. unihandle, multihandles, and omnihandles, extend from a SecureHandle, instead of a BaseHandle. The SecureHandle class also extends from

BaseHandle, which means that all secure M2MI handles are also BaseHandles, and are also SecureHandles.

- The original M2MI library had a HandleFactory class, which was used by the M2MI class to create handles. The HandleFactory would then use the HandleSynthesizer class for synthesizing actual handle classes.. A new SecureHandleFactory was created to create secure handles. This new SecureHandleFactory uses the SecureHandleSynthesizer to synthesize actual secure handle classes.

- A new interface, SessionKeyEstablisher, was added to support session key establishment. See Section 3.3 Session Key Establishment for a description of this interface.

- A new class, CipherSet, was added to implement interface SessionKeyEstablisher. This class handles the establishment of the session key. An instance of this class is created for each secure group name. It maintains the session key for the group, and refreshes that key when it expires. This class implements the protocol for establishing and maintaining a session key, and handles all of the protocol issues. It uses a few helper classes defined in the edu.rit.m2mi.security package.

- A new class, SessionKeyManager, was added to provide a mapping between a secure group name, and the CipherSet that encapsulates the session key for that secure group name. When the application wants to encrypt or decrypt a message, it will ask the SessionKeyManager for the CipherSet associated with the secure group name for the message. If none exist, a new one will be created. If one already has been created, it will be returned. It will then use the CipherSet object to encrypt or decrypt the message. When the secure M2MI layer is done encrypting/decrypting the message, it will notify the SessionKeyManager that it is done with the CipherSet.

6 Test Clients

To test the security features of this project, two of the existing M2MI test clients were modified. These are the Ping Pong program, and the Chat program. These changes, and what they test, are described in the following subsections.

6.1 Secure Ping Pong

- The original Ping Pong programs had a single interface, PingPong. It had two programs, Ping, and Pong, which implemented the PingPong interface. The Ping program would print out a “Ping” message every time it received the method invocation for ping(). It would then sleep 1 second, then invoke method pong() on an omnihandle that it created at startup. The Pong program would print out a “Pong” message every time it received the method invocation for pong(). It would then sleep 1 second, then invoke method ping() on an omnihandle that it created at startup. This is a simple test program for testing omnihandles.

- The Secure Ping Pong programs use the same PingPong interface, but create secure omnihandles, and export an object implementing the PingPong interface using secure M2MI.

- In addition to these simple test programs, two new programs were created: a PingMonitor program, and a SecurePingMonitor program. These two programs export interface PingPong, just like the Ping/Pong/SecurePing/SecurePong programs do. However, these programs are passive. They don’t respond to the incoming messages. Instead, they display all messages coming in, as follows. If a byte in the incoming message is in the range ‘ ‘ to ‘z’, it is displayed as a character. Otherwise, it is displayed in hex. For the standard `PingMonitor` program, three strings can be recognized in the output. These include the magic number “M2MI”, the interface name, and the method being invoked. Here is sample output from this test program:

```
< /127.0.0.1:5678 a386e3e180000M2MI9cebac200000000000000000001aedu.rit.
m2mi.test.PingPong04ping04(I)V000aaced05w40001dc8d
Received Ping 1
```

- For the `SecurePingMonitor`, two strings can be recognized, the magic number “M2MJ”, and the secure group name. The rest of the message is encrypted, and is not recognizable. This makes this program a good demo program for showing that the messages being transmitted are being encrypted. Here is sample output from this test program (this test client uses `myPingPongGroup` as the secure group name):

```
< /127.0.0.1:5678 1a-efa80000M2MJd183<8b0fmyPingPongGroup000‘#f1ob7c93
qac7ada24e51bd8&dadedd3d8d416be86b7c9dbd0FfeQ!9cd6a1Ua9afT_?7ae41F7bn9
3Ef4?N94pa8-W16<98ee1aaa97dd5Tbpe697ff1a0wdeec0b11bb911c8d8bd8dKY[fJs9
be5/81
Received Ping 1
```

Notice that even though the message dump only has two recognizable strings, the program was still able to decipher the message and display the correct output.

The secure M2MI code was written to retrieve the authentication key from a file. It defaults to a file in the user’s home directory called “auth_key”. The user may override this default behavior by defining a system property of “AUTH_KEY_FILE”, and specifying the path to the file containing the authentication key. This behavior is used to test Intruder Detection. The following procedure was used to test this:

1) A `SecurePing` program was started with authentication key A, group name G.
2) A `SecurePong` program was started with authentication key B, group name G.
3) A `SecurePong` program was started with authentication key A, group name G.
4) A `SecureStartPingPong` program was started with authentication key A, group name G.

The output of the programs that started with authentication key A all output the same results as if the `SecurePong` program using authentication key B wasn’t running. The `SecurePong` program using authentication key B displayed error messages when it tried to decrypt the session key received, even though it was using the same group name.

6.2 Secure Chat

The original Chat program had a single interface, Chat. It also had a single program named ChatDemo, which implemented a rudimentary chat program. This chat program allowed the user to type in a text message. When the user pressed enter, the message (and username) was broadcast to all objects implementing interface Chat. The ChatDemo program also exported an object that implemented the Chat interface, and displayed all messages received on that interface. The Chat program is discussed in the M2MI OOPSLA paper [KD02].

A secure Chat program was developed, which adds a little more functionality to the original functionality, besides adding encryption. In order to test multihandles, a test client needed to be developed that broadcast messages on multihandles. To do this, the secure chat program uses multihandles to send messages to each other. These multihandles are passed around among applications using the first ChatDiscovery interface defined in the M2MI OOPSLA paper [KD02] (the one that only has a report method, not the one that has request and report methods).

When the secure chat program starts, it takes as command line parameters a user name, a secure group name, and a session name. It uses the secure group name to create a secure omnihandle for interface ChatDiscovery. It uses the secure group name to export interface ChatDiscovery. It then waits up to 15 seconds for somebody to report a multihandle for the same session name it was started with. If a multihandle is received with the same session name, and a multihandle hasn't already been created/saved, the incoming multihandle is saved. If no multihandle is received with the same session name after the timeout period, a new multihandle is created and saved. When the new multihandle is created, the applications starts periodically invoking report on its omnihandle for interface ChatDiscovery. While this is not a complete implementation of the ChatDiscovery interface, it is sufficient for testing secure multihandles, which was the goal of this test client.

The rest of the secure chat program is the same as the original chat program. A user may enter messages, and when they press enter, the message is broadcast on the secure multihandle that was created/received at startup. Whenever a chat message is received on its securely exported Chat object, the message (if decrypted successfully) is displayed to the user.

Like the Secure Ping Pong program, a monitor program was written for the Secure Chat program. This monitor program is similar to the Secure Chat program, except instead of displaying messages to the user, it displays the raw message (in the same fashion that the Secure Ping Pong program does), and displays the text message sent as an argument to the method invoked on the secure multihandle.

7 Measurements

In order to ensure that this package will work on a small device, the following measurements were taken and evaluated.

7.1 Code Size

This measurement is used to determine the amount of memory required to store the M2MI code. The code size was measured before and after the security changes were added. The results are given in the table below.

Old Code Size	**New Code Size**
504832 Bytes	599040 Bytes

Table 1 - Code Size Deltas

The code size was increased by approximately 19%. While this is significant, it isn't overwhelming.

7.2 Session Key Memory Usage

This measurement determines the amount of RAM needed to store all key related structures. The actual memory used cannot be determined. The Java language does not provide a method for determining the size of an object. It does provide methods for serializing objects, provided that all of the objects are serializable. Serializing the object converts it to a byte array, whose length can be retrieved. However, some of the Java classes that were used were not serializable, so even the serialized size could not be determined. So, instead of providing the amount of memory used, this section describes the steps taken to minimize the amount of memory used.

The `CipherSet` class stores the session key, along with other fields required to maintain that session key. In order to minimize memory usage, a map was created that mapped a group name to the `CipherSet`, so that only one `CipherSet` is created per group name used. If an application exports multiple objects using the same group name, they will all use the same `CipherSet`, and the same session keys. If that same object creates a handle with the same group name, it will also use the same `CipherSet`.

Each `CipherSet` has 3 copies of the session key. One is stored in encrypted format so that any requests for the current session key can be answered without having to re-encrypt the session key. The other two copies are stored in the encrypt Cipher and the decrypt Cipher. These objects are required for performing the encryption and decryption routines. The number of copies of the session key could be reduced to 2 by not caching the encrypted session key. However, a performance hit would be taken whenever the session key needs to be reported, since the session key would then need to be encrypted with the large authentication key before sending.

7.3 Non-Volatile Memory Usage

This measurement determines the amount of NVRAM needed by this package. This includes the amount of memory needed to store the group authentication keys. The amount needed will vary from device to device, depending on how many groups the user of each device belongs to. This amount is also dependent on the authentication key size, which is hard-coded in this project, but could be easily adjusted.

This project uses `DESede` (a.k.a. Triple-DES). The group authentication key size used can vary. A good strength key is approximately 1024 bits, or 128 bytes. This key needs to be stored, as well as the group name to associate with it. The group name can vary greatly, and is not limited by the application. The total NVRAM needed would be based on the following calculation:

#groups * (authKeySize + average(groupNameLength))

For instance, if a user belonged to 20 groups, the authKeySize was 1024 bits, and the groupNameLength average size was 64 bytes, the total amount of storage required (if stored unencrypted) would be less than 4 KB.

The important factor here is that the user does not need to maintain a separate key for each user that they wish to communicate with. They only need to maintain a separate key for each group that they belong to, which is typically much smaller.

7.4 Session Key Generation Time

This measurement determines the amount of time required to initialize the security objects and to exchange messages in order to establish the initial session key. This time was measured with various numbers of clients wishing to join a communication group at the same time. This time will be called the startup time in this section.

For the first application to startup with a given session key, the startup time varied from 15-25 seconds.

Part of that time is a 5 second key wait time that the first application must wait before generating the session key.

For applications that start later, the startup times varied from 10 seconds to 15 seconds, and was not affected by the number of clients running. This makes sense, since the startup time is dependent on the time to complete initialization methods (fixed amount of time within the process) plus the time to send a request, that request to be processed by one receiver, and one response message. Since only one receiver responds, only one set of messages are exchanged.

Most of he variability in these timing measurements is explained by the use of a backoff timer. In order to prevent a flood of responses to key requests, a backoff timer was used, which ranges from 0 to 3 seconds.

7.5 Message Encryption & Decryption Time

This measurement determines the amount of time required to encrypt or decrypt a message. This time was measured with small and large messages on a Sun Ultrasparc workstation. Encryption and decryption times were approximately the same, +/- 1 msec of each other.

Message Size	**Encryption/Decryption Time**
10 Bytes	1 millisecond
10000 Bytes	20 milliseconds

Table 2 - Message Encryption and Decryption Time

For small packets, the throughput was about 10 Kbytes/sec. For large packets, the throughput was about 500 Kbytes/sec.

Most method invocations will be under 2 Kbytes. So, if the embedded device was 20 times slower than the workstation, it could encrypt and decrypt a 4 KB message at the rate of about 5 per second.

8 Future Work

This project has integrated encryption into M2MI, assuming that an authentication key has already been established for the device running secure M2MI. Work is still needed to establish the authentication key on the device, either from downloading from a host PC, or some other means. See [Bi03] for one way that this might be done (note that this reference is to a thesis that is work in progress).

Work needs to be done to store the group authentication key on the device in a secure way, so that hackers can’t view the authentication key/keys. The current implementation assumes that group authentication key is stored in a plaintext file.

As with the rest of M2MI, this project currently relies on the J2SE platform. In order to port this to an embedded device, a device profile needs to be created that supports some of the `javax.crypto` package, and other packages/classes used that are not available in the J2ME platform.

There is a small possibility that messages may be dropped when the session key expires, and new keys are being exchanged. The key exchange messages won’t be dropped, but clients using the old key will not be able to decrypt messages transmitted with the new key, and vice versa. This could be partially fixed by caching the previous session key, and if the decryption fails with the current key, attempt to decrypt with the previous key. This will fix the case where a client with a new key receives a message encrypted with the old key, but not vice versa. To fix the other case, where a message encrypted with a new key is received prior to receiving the new key, might be more difficult. In order to make some of the encryption and decryption methods thread safe, some of the methods were synchronized. This may prevent the key from being received before returning from the decrypt methods. This work is beyond the scope of this project, but would be a good area for enhancement.

9 References

[BdM93] Benaloh, Josh and de Mare, Michael. One Way Accumulators: A Decentralized Alternative to Digital Signatures. In *Advances in Cryptology (EUROCRYPT '93), Proceedings of the Workshop on he Theory and Application of Cryptoghapic Techniques*, pages 274-285, May 1993.

[Bi03] Binder, Joseph. Dynamic, Fault-Tolerant Key Management for Ad Hoc Networks. Master's Thesis, Jan 2003.

[Br97] Branchaud, Marc. A Survey of Public Key Infrastructure. March 1997.

[FFS87] Feige, Uriel, Fiat, Amos, and Shamir, Adi. Zero knowledge proofs of identity. In *Proceedings of the 19th ACM Symp. on Theory of Computing*, pages 210-217, May 1987.

[Ja96] Jablon, David P. Strong Password-Only Authenticated Key Exchange. ACM SIGCOMM Computer Communications Review, October 1996.

[KB02] Kaminski, Alan and Bischof, Hans-Peter. Many-to-Many Invocation: A New Object Oriented Paradigm for Ad Hoc Collaborative Systems. In Proceedings of the 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002), November 2002.

[KPT00] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. Proc. of 7th ACM Conference on Computer and Communications Security, pages 235–244, November 2000.

[MvOV96]Menezes, A., van Oorschot, P., Vanstone, S. Handbook of Applied Cryptography. CRC Press, 1996.

[NTCS99] Ni, S.-Y., Tseng, Y.-C., Chen, Y.-S., and Sheu, J.-P. The Broadcast Storm Problem in a Mobile Ad Hoc Network. In Proceeding of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99), pages 151-162, August 1999.

[STW00] M. Steiner, G. Tsudik and M. Waidner. Key Agreement in Dynamic Peer Groups. Presented at IEEE Transaction on Parallel and Distributed Systems, Aug. 2000

[STW96] Steiner, Michael, Tsudik, Gene, Waidner, Michael. Diffie-Hellman Key Distribution Extended to Group Communication. In ACM Symposium on Computer and Communication Security. March 1996.

[St95] Stinson, Douglas R. Cryptography: Theory and Practice. CRC Press, 1995.