Prototype Development for Secure Public Instant Messaging (IM) At Work*

Nigel Williams, Joanne Ly†
Centre for Advanced Internet Architectures. Technical Report 041123A
Swinburne University of Technology
Melbourne, Australia
{182329,182268}@swin.edu.au

Abstract – **This report details the development of a secure messaging add-on for use on public Instant Messaging (IM) networks. It is a continuation of the project discussed in** [25]. **Due to sociopolitical and monetary reasons, the growth of secure enterprise IM systems has not kept pace with the use of insecure public IM systems. A solution to this problem is to develop a free prototype that is able to secure communications over a public IM network, using a set of guidelines for information security and IM network compatibility. The prototype was able to successfully integrate with a third party IM client and conduct an encrypted session over the MSN Messenger Network. With more development and testing the prototype could become a viable option for use as a secure IM add-on.**

Keywords – **Instant Messaging, MSNP, OSCAR, encryption, information security**

TABLE OF CONTENTS

TABLE OF FIGURES

*This report was originally developed and submitted for 'HET556: Design & Development Project 2', Semester 2, 2004

†The authors are undergraduate students in Engineering at Swinburne University of Technology

TABLE OF TABLES

I. Introduction

Instant Messaging (IM) is an Internet-based application that allows for real-time communication between users as well as providing functions such as file transfers and video conferencing.

Once thought of as a time-wasting tool for sending personal messages at work, IM has evolved into a more practical application that provides an easy way to exchange files and conduct spontaneous online meetings. This evolution of IM applications has caused the IM industry to see a rapid growth in the number of users in recent years.

IDC research found in 2003 that there were over 43 million users of public IM in the workplace [1]. This number is indicative of how IM has involved from being a "virtual water cooler" [1] to an alternative business communication tool.

The major problem with IM usage in the workplace however is that the IM programs used are insecure public IM applications. A Group study conducted in 2003 found that nearly 80 percent of instant messaging in workplaces was done using public IM services.

A study conducted by AT&T Laboratories between 2000 and 2001 found that the majority of workplace IM interactions were "complex, work-specific interactions" [14]. This emphasises the need to make IM more secure if it is to be used in the workplace.

A. Background

There are two types of IM applications available to users: public and enterprise. Public applications are downloadable off the Internet for free. They are the most widely used type of IM application in both the home and workplace. However, in the workplace, they pose a major security risk to an organisation. This is mainly due to the fact that the data that is exchanged over public IMs can be intercepted and distributed. If the data intercepted was sensitive, this could have a catastrophic effect on an organisation.

Enterprise IM applications were developed to combat the security issues that public IM applications posed on organisations. Enterprise IM applications are not free and must be purchased as with any proprietary software package.

The initial aim of the project was to build a secure IM application that could be used in the workplace. However, after further investigation into the reasons why enterprise solutions were not as commonly used as their public counterparts, it was discovered that the issue was of a sociopolitical nature than a technical one.

Looking at the employee and manger attitudes, the following were some of the reasons deduced to explain why enterprise IMs were not widely used [25]:

- Some managers did not see the purpose in implementing IM at work
- It may be difficult for managers to find a solution to deal with IM usage – should it be banned altogether or if not, what kind of restrictions should be imposed?
- Some employees may resist being restricted to using enterprise IMs as could cut off their ability to communicate with people outside of work
- Companies may not have the monetary resources to purchase an enterprise IM system
- Companies may not have any available IT resources to manage the system

From this, it was decided that a more constructive approach would be to try and make public IMs more secure whilst they were still more predominately used than enterprise IMs. To do this, the aim of the project changed to make an encryption add-on in an attempt to secure public IMs at work.

The next phase of the project was to develop a proof of concept prototype using DES encryption for a Java-based, open-source Microsoft® MSN Messenger (or simply, MSN) IM client called TjMSN. The prototype proved that public IMs could easily be made secure by using a simple add-on.

For this semester's work, as the prototype was a simple demonstration, it needed to be further developed to include such information security objectives such as authentication, data integrity and key management to become a more viable solution. In addition to that, DES is an old encryption algorithm that is succeeded by more efficient, newer algorithms. Therefore, the algorithm was to be replaced with one of the newer algorithms.

B. Motivation

Public IM applications are increasingly being used in the workplace. The Telematica Institut in the Netherlands conducted a study in 2004 to investigate how public IM applications were adopted in the workplace [3]. The study found that the usage of IM increased fourfold, in both users and conversations, after the formal introduction of IM into the workplace.

The increase in public IM usage and the lack of management control induces security risks within the workplace. The project motivation was to devise and interim solution to the security risk whilst public IMs were increasing in usage without any management control.

C. Context of Use

The context of use for which the prototype was designed for was based on the last two points listed previously as reasons why enterprise IMs were not extensively used (lack of monetary funds and IT resources). The following describe the context of use for the prototype:

- Small company (no more than 100 employees)

- Lack of monetary funds to purchase enterprise IM
- Lack of IT resources to manage an enterprise IM system
- Employees that see the benefits of using public IMs but do not wish to jeopardise the company by having their IM exchanges intercepted by adversaries

Therefore, the prototype needed to be free, secure, easy to use and not take up too much IT resources.

D. Outcomes

The prototype was developed with two major design focuses: information security principles (VI) and IM design principles (VII). Information security principles are concerned with how the information that is being exchange is kept secure from eavesdroppers. This included the application of cryptography to keep messages secret, authenticate entities and implementation of a secure key management exchange.

Considerations were also made in regards to the context of use in which these information security principles will be implemented. IMs function in real-time, so it was imperative that the prototype did not slow down the communication excessively. IMs also have a message length limit (payload). This created a restriction on the length of the encrypted text that was to be exchanged over the communication channel. Thus, tradeoffs between information security principles and IM design principles were made to develop a secure add-on that would be effective for an IM application.

Once the prototype was developed, it was tested in a small network environment on different operating systems and processors. The outcome of the tests found that the prototype did not adversely affect the real-time attribute of IMs when providing information security measures.

The cryptographic world is constantly evolving. During the late stages of development, it was found that some cryptographic primitives were being included in code libraries that were more effective than the ones chosen for the prototype. Therefore, it is recommended that the prototype should be developed further before being implemented in the “real world”.

II. Objectives and Scope

A. Objectives

It was found in the previous semester's work [25] that although there were many enterprise IM solutions available, the problem with their lack of use had to do with manager and employee attitude. Therefore, it was decided to change the original objective of designing a secure IM application to designing a cheap, workable add-on that will try and solve IM security problems whilst public IMs were still predominately used.

The objective for the project was to develop an encryption add-on for an MSNP (MSN Messenger Protocol) based [25] based client that met the information security objectives in (VI). Although there are encryption add-ons that are currently available for download (IV), the add-on to be developed for the project had a small workplace focus and was also tested under similar conditions to that of a small workplace.

B. Scope

The prototype that was developed for the proof of concept in the previous semester was a simple design that would be impractical to implement in a real world environment. The purpose for developing the prototype was to develop a familiarity with how to implement a cryptographic algorithm for a specified purpose.

During the second stage of project development, the following was done to develop the prototype into a feasible workplace solution:

- Implement a more effective algorithm than DES to do the encryption of the messages sent
- Provide a key management system required to ensure that the keys used for encryption will distributed appropriately
- Provide a data integrity mechanism to ensure that messages aren't tampered with by a third-party
- Provide an authentication mechanism to ensure that only authorised parties are privy to sensitive information

When developing these features, considerations were made towards:

- The level of security that would be provided
- The performance of the add-on. The add-on must be practical and not inconvenience users.

Originally, once the MSN (MSNP based) prototype had been developed into a feasible add-on, it was planned to attempt to adapt the code for an OSCAR (Open System for Communication in Real-time) based IM. OSCAR is the protocol used by ICQ and AIM. However, during the beginning of the second stage of development, it was found that Microsoft® had altered the method of accessing the login servers. This meant that TjMSN had a major release update that needed to be studied to ascertain how the prototype was to be affected. This meant that the project fell behind schedule and an actual OSCAR implementation was not developed. However, the prototype did not include any MSN specific functionality. Therefore, the prototype would need only minor changes to accommodate an OSCAR implementation.

III. MAJOR CONCEPTS

This section describes the major theories and concepts that formed the backbone of the project. These concepts were studied in depth during the first semester's development work [25] and were used as a reference during the second semester's project development.

A. IM Protocols

IM protocols define the way an IM application functions, including the services they provide, the network components needed and how these components interact with each other. Currently, the most widely used public IM applications in the workplace [2] are:

1. AOL Instant Messenger (AIM)
2. Microsoft MSN Messenger
3. Yahoo! Messenger
4. ICQ

Both AIM and ICQ use the Open System for Communication in Real-time (OSCAR) protocol. Although both IMs use the same protocol, they are not interoperable. MSN uses the MSN Messenger Protocol (MSNP) and is also the protocol used for the project prototype.

Understanding the IM protocol is important as it details what the IM application is capable of and what sort of design limitations may be imposed. For an in depth discussion of OSCAR and MSNP, refer to [25].

B. Cryptographic Basics

In a general scenario involving communication between two entities over an insecure channel, the communication exchange runs the risk of being intercepted by an unauthorised entity. This scenario is shown in (Figure 1 - Basic Communication Exchange). Alice and Bob exchange the message *m* with each over the insecure channel. Eve is listening in on the exchange and also receives *m*.

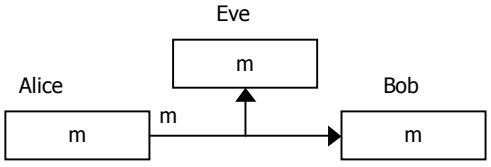

Figure 1 - Basic Communication Exchange

If Alice and Bob were not exchanging sensitive data, Eve's eavesdropping would be a harmless nuisance. However, if the data exchanged was sensitive and Eve was able to intercept it, then the result may be disastrous.

Cryptology is the technology used to keep sensitive data, exchanged between two parties, hidden from eavesdroppers. This is done by encrypting the data being exchanged in a manner that only the intended receiver can decrypt it.

Referring back to the general scenario, encryption will now be used in the communication exchange between Alice and Bob. First, Alice and Bob agree on a secret key K_e. This will be done via another communication channel that Eve cannot intercept. For example, Alice can mail Bob the key.

When Alice goes to send the plaintext message *m*, she encrypts it using an encryption function *E* and the key. This results in a ciphertext *c*, which is sent to Bob. As Bob knows the secret key, when he receives *c*, he is able to decrypt with the decryption function *D*. The result of this function is the message *m*, which Bob can now read. This process is shown in (Figure 2 - Generic Encryption Scenario). Eve can still intercept messages sent on the channel, however now Eve intercepts *c* instead of *m*. As Eve does not know what the key K_e is, Eve cannot decrypt the ciphertext and therefore cannot read the message sent.

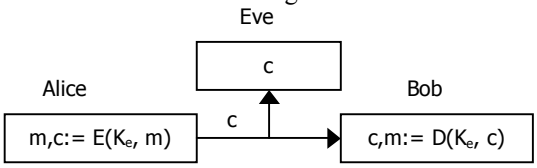

Figure 2 - Generic Encryption Scenario

C. Cryptographic Algorithms

As seen in the generic encryption scenario above, Alice uses an encryption function to convert the plaintext message into ciphertext. The function uses a cryptographic algorithm to perform the conversion. There are many different cryptographic algorithms. The algorithms can be grouped into classes called primitives. There are three main primitives: unkeyed, symmetric keyed and asymmetric keyed (Cryptographic Primitives).

Unkeyed algorithms do not require any keys. An example includes arbitrary length hash functions, the main subclass of unkeyed algorithms. Hash functions work by generating a hash value (a small message digest) from a large message source. Hash functions are implemented in many cryptographic processes, including digital signatures, key establishment and random number generations. Hash functions are also implemented in Message Authentication Codes (MAC). However, as MACs use a symmetric key, they are therefore classed under the symmetric keyed primitive.

Symmetric key algorithms are also known as secret key algorithms. These algorithms use a single key, called a secret key, to manipulate the data. The secret key is shared by authorises entities and is kept secret from everyone else. Symmetric key algorithms can be implemented in the following ways:

- To provide confidentiality by using the same key to encrypt and decrypt data. Unauthorised entities should not know this key.
- Generation of pseudorandom numbers
- Form part of a key establishment process
- Perform authentication and data integrity checks in the MAC process through using the same key to generate and validate the MAC

Asymmetric key algorithms are also known as public key algorithms. These algorithms use a set of two related keys known as a key pair. Both authorised and unauthorised entities are able to know what the public key is. The private key should only be known to the entity that owns the key pair. The relation between the private and public key is such that the private key cannot be discovered using the public key. Asymmetric key algorithms are implemented in the following way:

- Calculating digital signatures
- Establishing cryptographic keying material

Each cryptographic algorithm should be evaluated according to the following criteria to assess which is the best algorithm for the required functionality:

1. *Security level:* Usually, this criterion is difficult to determine. Normally, the level of security is defined by the upper bound of the level of work necessary to defeat the targeted information objective/s. This upper bound is also known as the work factor.

2. *Functionality:* As stated previously, a combination of cryptographic algorithms is needed to cater for all four of the information security objectives. The functionality defines which objective/s the algorithms are the most suitable to.

3. *Modes of Operation:* The algorithms will display varying characteristics depending on how they are applied and what type of inputs used. Therefore, the mode of operation will determine what type of functionality the algorithm will provide.

4. *Performance:* This measures the efficiency of an algorithm, depending on its mode of operation.

5. *Implementation Ease:* This measures how easy it is to implement the algorithm in a practical environment. For example, sometimes the environment may cause there to be a performance or level of security trade off because the hardware or software may be below the minimal requirements needed to run a more efficient or secure algorithm.

D. Information Security

The main concept behind cryptology is information security. Information security aims to handle and minimise data communication problems. Not only can sensitive data be read by eavesdroppers, the data can also be intercepted, altered and sent to the intended receiver or passed onto an adversary. Another data communication problem is disputes about the content of a past communication exchange.

To deal with these problems, information security has the following main objectives:

- **Confidentiality (or privacy):** concerned with preventing eavesdroppers from being able to read the information that they have intercepted.
- **Data integrity:** involves ensuring that the data received has not been altered in any way. Alterations include deleting and modifying all or part of the data as well as inserting additional data.
- **Authentication:** focused on establishing the identity of the user or system that originated the data.
- **Non-repudiation:** aims to prevent the denial of previous commitments or actions as well as prove the integrity and origin of the information independently by a third party.

In general, most data security systems combine two or more of the above objectives to gain a satisfactory level of security. For example, confidentiality would not ensure that the entity that the communication exchange is being conducted with is an authorised entity. Hence, authentication needs to be implemented in conjunction with confidentiality.

One of the aims of the project was to include all these security objectives. Some cryptographic algorithms are able to serve multiple objectives; such as digital signature algorithms are able to provide authentication, data integrity and non-repudiation.

E. Kerckhoff's Principles

August Kerchoffs, one of the great early cryptologists, published in 1883 what he deemed were the six essential attributes that a military cipher should contain. They are [7]:

- The system should be, if not theoretically unbreakable, unbreakable in practice.
- Compromise of the system should not inconvenience the correspondents.
- The key should be rememberable without notes and should be easily changeable.
- The cryptograms should be transmissable by telegraph.
- The apparatus or documents should be portable and operable by a single person.
- The system should be easy requiring neither knowledge of a long list of rules nor involving mental strain.

With modifications to adapt to the modern computer world, these attributes were taken into consideration when designing the project.

The second attribute was later expanded and became known as 'Kerckhoffs' Principle':

"The security of the encryption scheme must depend entirely on the secrecy of the key and not the secrecy of the algorithm." [7] This principle formed an important basis for the design of the project.

IV. CURRENT SOLUTIONS

Through the second semester of the project development, investigations into what solutions were already available were made periodically. The below are the most recent solutions that were looked at. These solutions were also discussed in the context of usage defined for the project.

A. CryptoHeaven

CryptoHeaven [26] is an application that offers a secure Internet communications service. It consists of the following:

- Secure email
- Secure online storage, file sharing and distribution
- Secure instant messaging
- Secure and private discussion forums

The features above are integrated into a single user interface. There are two types of accounts offered: free and premium.

The cryptography used includes the Advanced Encryption Standard (AES) cipher with a 256-bit symmetric key, public key cryptography using 2048 to 4096-bit asymmetric keys and the Secure Hash Algorithm (SHA-256) message digest function.

The major difference between this product and the solution for the project was that the product offered a secure solution to numerous Internet communication services such as email and instant messaging. The project's focus was solely on securing instant messaging. Therefore, certain design considerations may be different to that of CryptoHeaven.

B. Top Secret Messenger (TSM)

Top Secret Messenger (TSM) [27] is a public key encryption plug-in for popular IMs and e-mail clients such as ICQ Instant Messenger®, MSN and Microsoft Outlook®. TSM uses Elliptic Curve Cryptography (ECC) algorithm as well as SHA-1 hash function and Triple DES (3DES).

There are two versions of the plug-in available: trial and registered. The trial version is basically a demo of TSM technology. It uses weak 8-bit encryption keys instead of the 307-bit keys used in the registered version.

Although the product provides a solution to the problem investigated for the project, it is at a cost. It was decided in the first semester [25] that the focus would be on small business that did not have the monetary funds to buy enterprise IMs. In the free version of TSM, the security provided is minimal and not feasible for actual workplace usage.

C. Secway Simp Pro and SimpLite

Similar in concept to the encryption add-on proposed by the group, Simp Pro targets the corporate environment and provides encryption of conversations across a number of popular IM networks with a variety of encryption algorithms. It also has the ability to encrypt file transfers in ICQ and MSN.

Public Key ciphers supported are RSA (2048 to 4096-bit), Diffie-Hellman, DSA and ECC. Symmetric ciphers available are AES, 3DES, CAST and Twofish, all of which are limited to 128-bit key lengths.

SimpLite, a reduced feature version of Simp Pro, is available free for personal use. It supports two 2048-bit RSA keys per installation and can use either 128-bit AES or Twofish symmetric ciphers.

The cost of Simp Pro is 25€ per licence, or 45€ for two licences for small offices/home. Bulk purchases for enterprise users range from 10€ per licence starting from 10 licences.

V. Project Development Process

This section outlines the project development undertaken to design and develop the prototype. The development occurred in three major stages: prototype assessment, prototype design and prototype testing.

A. Prototype Assessment

The prototype developed in the previous semester was assessed to establish what parts of the prototype was reusable or needed further development. The majority of the assessment was related to the cryptography used and information security objectives. The old prototype only provided confidentiality. It did not have any authentication, data integrity or non-repudiation mechanisms. Research was done to gain knowledge on how to implement these mechanisms with respect to the application context.

Also, the encryption algorithm used (DES) was not a feasible solution as there are known attacks against it. This meant that a newer, more efficient algorithm needed to be investigated and implemented. During the assessment, considerations were also made in regards to the limitations that the context of use (i.e. in an IM application) were made to ensure that the final solution would be feasible.

B. Prototype Design

This section outlines the software development life cycle, development tools and cryptographic tools used to develop the prototype design.

1. Software Development Life Cycle

An evolutionary prototyping approach was taken during the design and development of the prototype. This was an iterative approach, refining parts of the system for each iteration. This approach allowed the reuse of code, including code from the original prototype developed in the first semester.

2. Development Tools

Crimson Editor (Windows) and Mi (OS X) text editors were initially used when developing the prototype. These tools were replaced, though, when a change in the MSNP login process required that code be transferred to a more recent version of TjMSN. The new development tool chosen was NetBeans IDE.

a) NetBeans IDE 3.6

NetBeans IDE is a free, open-source Java development environment available on a number of platforms. It is also the environment in which TjMSN was originally developed. NetBeans was chosen as the replacement environment as it allowed for easier editing of TjMSN's existing GUI and simplified compilation of classes with multiple dependencies. NetBeans also allowed the group to standardise development environments.

b) Java 2 SDK

The Sun Java 2 SDK1.4.2 was used to compile the Java code, which was the most recent SDK available at the time of development. Although Java 2 SDK 5 has now been released, the final implementation of the prototype was compiled using J2SDK v1.4.2.

c) Ethereal

The Ethereal packet analysis tool [18] was used to observe IM conversations as they appeared on the network before and after encryption was activated.

3. Cryptographic Tools

The J2SDK v1.4.2 by default does not contain the necessary packages or settings required to obtain a sufficiently high level of cryptographic security. The Java Cryptography Provider for Java Cryptography Extensions (JCE) supplied with v1.4.2 does not support the RSA algorithm, while the Security Policy file of the default Java installation prevents the use of long encryption keys. To reach a suitable level of security, a third party Cryptographic Provider and an enhanced Security Policy were required.

a) The Legion of Bouncy Castle

The Legion of Bouncy Castle [15] is a group who have written a set of cryptographic APIs for Java including a Cryptographic Provider. These files are available free of charge and are open source. The Bouncy Castle Cryptographic Provider supports a wide range of cryptographic methods and algorithms and was used in place of the Sun Java Cryptographic Provider for all functions of the prototype.

b) Unlimited Strength Security Policies

United States policy restricts the exportation of high strength encryption technology. Therefore, encryption key sizes for a default installation of the Sun JRE are limited to a "strong" length of 512-bits. As the RSA keys to be used by the prototype range up from 1,024-bits, it was required that a replacement "unlimited" strength US Export Policy File be obtained from the Sun JCE website.

c) Java KeyTool

An individual's private key, used to decrypt received messages and to sign outgoing messages, must be prevented from being accessed by any unauthorized persons.

Rather than create a program or method to protect and organise a user's keys, it was decided that an existing alternative should be located and used by the prototype.

The Java KeyTool [20] provides a method of generating and storing an individual's private key, in addition to any public keys they may have obtained, in single file called a KeyStore. This KeyStore is protected by a password and encrypted on the hard disk. Also, KeyStore files can be interfaced directly from within a Java program using the

methods provided within the Java Runtime Environment (JRE).

KeyTool was used to manage the key-pairs and public keys used by the prototype.

C. *Prototype Testing*

The prototype was tested often during development. Individual routines were first tested as standalone programs run from the console using a 'test harness' program. A test harness is a simple program which can call methods from within a class to test its operation, and does not require the TjMSN client to be running.

Once successfully tested, methods were written into the prototype classes and tested in a 'live' situation on the MSN Network.

a) *Testing Environment*

During early development the prototype was tested by running two instances of TjMSN on a single computer simultaneously. A connection to the MSN Network would be established and messages sent between the two clients.

Once the prototype had matured to a more complete form, several tests were carried out in an environment similar to that which might be found in a small company. This was a small LAN consisting of three Windows-based computers connected to the Internet through a gateway. Tests were also carried out by communicating between the LAN and a remotely located computer.

b) *Functionality Testing*

The prototype functions were tested by establishing and undertaking encrypted conversations between two clients connected to the MSN network. A debugging console was used during this process, allowing any exceptions or errors encountered to be traced. The code was also tested against a range of incorrect user input to aid in developing error-handling code.

c) *Performance Testing*

Test harness programs were written to interface with the prototype's encryption code and time the implemented functions on a number of different systems. This was done to observe the delay introduced into a conversation by activating encryption and establish a baseline system for acceptable performance.

VI. Information Security Design Principles

This section describes the information security design principles that were used to guide the development of the prototype. The actual design implementation of these principles, can be seen in section VIII.

The information security objectives (confidentiality, data integrity, authentication and non-repudiation) were used to form a set of requirements for the prototype. The aim was to ensure that the prototype employed these objectives as part of its design.

Another design focus was implementing Kerckhoffs' principle, whereby the strength of the encryption depended on the secrecy of the key. To achieve this, the appropriate key establishment and key management techniques needed to be implemented.

A. Confidentiality

The first information security objective dealt with was confidentiality. As seen in the generic communication scenario, the basic concern when exchanging sensitive data over a channel is that an unauthorised entity is able to intercept and read the data that is being exchanged. Confidentiality is concerned with preventing the ability of the unauthorised entity to read the data being exchanged.

The method to keep data confidential is to encrypt the data. There are two types of encryption algorithms that are normally used to do this: stream ciphers and block ciphers. Stream ciphers encrypt individual characters of a plaintext message one at a time. In contrast, block ciphers simultaneously encrypt groups (i.e. blocks) of characters of the plaintext message.

For the prototype, a block cipher was chosen to implement confidentiality. The major reason for this was that although there are plenty of resources for the theory of stream ciphers and stream cipher design principles, most stream ciphers are proprietary. For example, LEVIATHAN has a Cisco patent pending. Therefore, there are no official, standardised documentation or code libraries for the stream ciphers currently being used in systems.

The chosen block cipher was the Advanced Encryption Standard (AES), which is a US government standard cipher. The rationale behind this was that as a government standard, it had been proven to work effectively and had yet to have an effective attack against it. Also, because AES is standardised, all encryption libraries contain it, making the actual implementation of AES easier.

A key problem with using block ciphers is that the plaintext to be encrypted may be longer than the block length. To deal with this, a block cipher mode was used. The chosen block cipher mode was the Cipher Block Chaining (CBC) mode. CBC is the most widely used block cipher mode in current systems. Hence, like with AES, there were many documentation and code library sources to examine and choose from.

1. Block Ciphers

Block ciphers are a fundamental element in many cryptographic systems and belong under the symmetric key primitive. A block cipher is an encryption function that maps n-bit plaintext message blocks to n-bit ciphertext blocks. In other words, it is used for fix-sized blocks, with n being the block length. For short messages, the block cipher can be used directly. More commonly, however, the message length is longer than the block length. If this is the case, a block cipher mode should be used. The main reason why the cipher text generated is because this avoids data expansion.

Block ciphers are normally implemented with a fixed, secret key. This corresponds with Kerckhoff's principle, whereby confidentiality is dependent on the secrecy of the key as it is assumed that the algorithms for the encryption and decryption are publicly known. The key is a string of bits, like the plaintext and ciphertext. The key is chosen at random. The common key sizes are 128 and 256 bits.

For unique decryption of the ciphertext, the algorithm needs to be one-to-one, where the plaintext can be mapped to the ciphertext. For any fixed key, a lookup table that maps the plaintext to the ciphertext can be computed. The size of the lookup table would be huge. For example, for a block cipher with a 32-bit block length, the lookup table would be 16Gb [6].

Block ciphers on their own are used to encrypt information, providing confidentiality. As a building block in a cryptographic system, block ciphers can be used for a variety of functions. For example, as part of pseudorandom number generators, stream ciphers, MACs and hash functions, message authentication and data integrity techniques, entity authentication protocols and digital signature schemes.

When implementing block ciphers in practical applications, tradeoffs need to be made. These include speed requirements, memory limitations and platform restraints (e.g. hardware, software). This results in a tradeoff between efficiency and security. It is therefore best to look at several block ciphers before deciding which is the best for the intended application.

a) Advanced Encryption Standard (AES)

The block cipher that was chosen for this project was the Advanced Encryption Standard (AES) (Advanced Encryption Standard (AES)). AES was developed through the National Institute of Standards and Technology (NIST), who had asked for cipher proposals from the cryptographic community. The cipher that was chosen to become AES was the Rijndael algorithm and became a US government standard.

The Rijndael algorithm is a symmetric block cipher that processes data of 128-bit block length, using keys of either 128, 192 or 256 bits in size. Although Rijndael is able to handle additional block sizes and key lengths, this is not included in the NIST standard [4].

The structure of a single round of AES is shown in Advanced Encryption Standard (AES). Depending on the key size, the full encryption process consists of 10-14 rounds. The plaintext, which is 128 bits (16 bytes) in size, is input through the top of the structure. The plaintext is then XOR with 16 bytes of the round key.

Each of the 16 bytes is then used as an index into an S-box table. The S-box is a substitution box, which is a lookup table that is publicly known. All the S-boxes are identical. The S-box table maps 8-bit inputs to 8-bit outputs.

The bytes are then rearranged in a specific order and are mixed into groups of four using a linear mixing function, completing a single round. The term linear refers to the fact that each output bit of the mixing function is the result of the XOR of the input bits.

The core reason for why AES was the chosen block cipher was that it is a current US government standard. The cipher has yet to have an attack defined for it (although it may be attacked in the future) and is used extensively. In addition, it is relatively easy to implement and is supported by all cryptography libraries.

b) Alternative Block Ciphers

There were two main alternative block ciphers that were looked at: the International Data Encryption Algorithm (IDEA) and Triple DES (3DES). Both of these algorithms are based on the Feistel cipher. The Feistel cipher takes the plaintext and splits it into two halves, L and R. For each round, there's a subkey K_i, which is derived from the cipher key K. Within each round, L is XOR with $F(K_i, R)$, where F is some kind of function. L and R are then swapped. The main advantage of using a Feistel cipher is that for decryption, the same process is used. This makes it easier to implement.

IDEA [6] uses a 128-bit key to encrypt a 64-bit plaintext message into a 64-bit ciphertext block. The structure of IDEA is based on the Feistel cipher, whereby it uses 8 rounds with six 16-bit subkeys followed by an output transformation. The main reasons why IDEA was not used were:

- There are known attacks on IDEA, which have continued to improve [5]
- The block size is too small

3DES [6] is derived from the NIST standardised block cipher, the Data Encryption Standard (DES). DES was developed in the mid 1970s and was the first commercial-grade algorithm that had open and fully specified implementation documentation. Although it is the most well known symmetric key cipher, it is no longer useful in modern implementations. DES uses a 56-bit key and generates 64-bit ciphertext blocks. By today's standards, the key and block size are too small. 3DES uses three DES encryptions in sequence. This solves the small key size problem, however there is no known solution for the small block size problem. Additionally, DES is already a slow cipher by today's standards and 3DES is a third of the speed of DES. Therefore, 3DES was not chosen as the block cipher to implement in the prototype.

c) Key Size

AES can operate with key sizes of 128, 192 and 256 bits. Although a 128-bit key is sufficient for most applications, it is liable to collision attacks (Collision Attacks). This type of attack depends on the fact that collisions (duplicate values) appear more regularly than expected.

For example, the same key might be reused after exhausting all other key values for a particular application such as secure online shopping transactions. An attacker might expect this and be able to insert messages from the old transaction while the new transaction is occurring.

A recommended design rule [5] is: *"For a security level of n bits, every cryptographic value should be at least 2n bits long"*. Therefore, for 128-bit security, a 256-bit key was implemented. AES operates slower with a 256-bit key than a 128-bit key. However, for the intended implementation, the delay would be negligible to the user.

2. Block Cipher Modes

Block ciphers can only encrypt fixed-sized blocks. For encrypting messages that are longer than the block length, a block cipher mode need to be used. A block cipher mode is an encryption function built using a block cipher.

A major point to emphasise is that encryption modes are only able to prevent an eavesdropper from reading the data. There is no authentication mechanism. Thus, the eavesdropper is able to alter the data without needing to be able to read it. In a lot of situations, the damage caused by modified data is greater than the fact that the data is being read. Therefore, the encryption should always be combined with authentication.

The encryption and authentication method is still not entirely secure as the attacker will still be able to perform a traffic analysis. A traffic analysis involves determining the fact that a communication exchange is currently occurring, when it is occurring, how much data is being communicated and whom the communication exchange is with. Although traffic analysis can be prevented, it generally takes up a lot of bandwidth for general purposes and is therefore not implemented.

a) Cipher Block Chaining (CBC)

The most commonly used block cipher mode is cipher block chaining (CBC). This was main reason why CBC was the chosen block cipher mode as there were ample documentation sources and code library providers.

CBC uses an n-bit initialisation vector (IV). The inputs of the algorithm are the key, the IV and one n-bit plaintext block. Each block of the message is encrypted separately, with the plaintext first being XOR with the previous ciphertext block. This is referred to as the chaining mechanism. Also, the IV needs to be changed (using either a counter or random field) for each block. The reason why this is done is to ensure that if two plaintext blocks were the same, their ciphertext blocks would not be identical, reducing the amount of information given to the attacker.

b) Electronic Codebook (ECB)

An alternative mode that was looked at was the electronic codebook (ECB) mode [5]. Each block of the message is encrypted separately. This creates a problem in that if two plaintext message blocks are the same, then their ciphertext block will be identical, providing an attacker with information for cryptanalysis. For this reason, ECB was not selected.

c) Nonce-Generated IV

There are several methods of generating the IV used in CBC. These include fixed, counter, random and nonce generated (Initialisation Vector (IV) Generation Methods). The choice of how the IV is generated is important, as it determines how much information is given to the attacker.

For example, when using a counter generated IV, if the first blocks of a message have minor differences then the IV counter could possibly cancel the differences during the XOR process and generate identical ciphertext blocks. This gives an attacker enough information to draw conclusions about the differences between the two messages, which is highly undesirable.

A nonce (*n*umber used *once*) generated IV has the ability to deal with the problems in generating IVs. Each message is given a unique number called a nonce. The uniqueness of a nonce is its most important characteristic and the same nonce should not be used with the same key.

The nonce can be randomly generated or be a message sequence number. As discussed in section, message numbering helps ascertain whether or not an adversary has deleted or inserted messages during a communication exchange. Therefore, the nonce that was used in the prototype also had a dual role as the message sequence number.

B. Data Integrity

Data integrity is concerned with ensuring that the data being transmitted is not altered in any way during its path through the communication channel, to the intended recipient. Different types of alterations can occur, including deleting and modifying all or part of the data as well as inserting additional data.

Looking back at the generic encryption scenario, when Alice sends the message *m*, Eve alters the message to *m**. So, Bob receives *m**instead of *m*. When Bob receives a message, Bob needs to determine whether the message was the one that Alice sent. Therefore, Bob should not assume that all messages that he receives comes from Alice. However, if he does not know who sent the message, then the message exchange is useless. Subsequently, to help with ensuring data integrity, authentication should be implemented so that Bob knows whether the message came from Alice or not. This type of authentication is known as message authentication (or data origin authentication).

Nonetheless, having authentication only does not completely solve the problem of data integrity. Eve is still able to delete messages, insert old messages or change the order of the messages. Therefore, message numbering should also be implemented so that Bob knows that he is getting the correct sequence of messages.

The message sequence numbering policy used for the prototype was that the sequence number started at zero and incremented sequentially by one for each successive message. If Bob receives a message whereby the sequence number has not been used previously in the current exchange and satisfied the condition that it was one greater than the previous number, the message would be accepted.

Another method to ensure data integrity is for Alice to generate a message digest using a hash function [3] and sends it to Bob along with the ciphertext message. When Bob receives the message, he also takes generates a message digest and compares it the one Alice sent. If the message digests match, it confirms that the message Alice has sent has not been altered in any way.

The primary reason why the latter solution was not implemented was by sending a hash along with the message, it may exceed the message. There are ways to counter this, however it would make implementation more complicated. Therefore, the first method of ensuring data integrity was used, with the details of authentication illustrated in section (Message Authentication).

C. Authentication

The aim of authentication is to be able to establish the identity of the entity that originated the data. This is important as it ensures the data sent is coming from a trusted, authorised entity. Essentially, there are two types of authentication: entity authentication and message authentication.

1. Entity Authentication

Entity authentication is concerned with the verification of an entity's identity in real-time, while the entity waits.

The entity that is questioning the identity is known as the *verifier* and the entity whose identity is being questioned is known as the *claimant*. The most common method of entity authentication is for the verifier to challenge the correctness of a message by checking to see whether the claimant is privy to a secret that is associated with an authorised entity. This is known as challenge-response identification.

a) Challenge-response Identification

The claimant confirms its identity to the verifier by demonstrating that they are privy to a secret associated with the verifier, without actually revealing the secret itself to the verifier. The "challenge" is normally a number that has been chosen secretly by one entity at the beginning of the identification protocol.

As well as operating in real time, entity authentication also makes numerous challenges throughout the communication exchange to ensure that an adversary has not "hijacked" the communication exchange. Given that there are numerous entity authentication attempts during a communication exchange, the challenge should be a time-variant parameter that is always unique. A nonce satisfies this requirement and is therefore used as the challenge.

As stated in section VI.A.2.c), the nonce used for the prototype is also a message sequence number. In the context of entity authentication, the sequence number can be used as a challenge as it is specific to a particular pair of entities and must be explicitly or implicitly associated with both the verifier and the claimant.

(1) Challenge-response Using Symmetric Key Encryption

The challenge-response mechanism used for the prototype involved symmetric (secret) keys. This required the verifier and the claimant to share a symmetric key. How the key is established between the two entities is detailed in section VI.E - Key Establishment and Management. The claimant uses the key to encrypt the sequence number (the challenge) thereby demonstrating the knowledge of the secret key and the challenge, proving the claimant's identity.

The reasons why this technique was used for entity authentication were that the parameters used during the authentication (the message sequence number and the secret key) were also implemented for other functions within the prototype. This created fewer overheads for the prototype, which was desirable with respect to the context of use of the prototype.

2. Message Authentication

Message authentication is also known as data origin authentication. It is closely related to data integrity as it checks the data origin, making sure it came from a trusted source. Data that has been altered has a new source. For example the data that Alice sends has been altered by Eve, making Eve the new source of the data. As mentioned in the data integrity section (VI.B), Bob should not assume that Alice sent the message. If Bob cannot determine who sent the message, then the message itself is useless. Hence, message authentication essentially provides data integrity and vice versa.

Methods of providing message authentication include [6]:

- Message Authentication Codes (MACs)
- Digital signature schemes
- Before encryption, appending a secret authenticator value to the text to be encrypted

It needs to be noted that unlike entity authentication, message authentication does not operate in real-time, as it does not have any guarantee of when the message was created.

3. Message Authentication Code (MAC)

Message Authentication Codes (MACs) are used to provide data origin (or message) authentication. They are a special type of hash function where one of its inputs is a secret key. The MAC function outputs a MAC value that is sent along with the encrypted message. The receiver generates a MAC value of the encrypted message received and checks to see whether it is the same as the MAC value received. The MAC values would be equal if the message has not been tampered with.

a) CBC-MAC

CBC-MAC is an algorithm that converts a block cipher into a MAC, using the secret key of the block cipher. CBC-MAC involves encrypting the plaintext message using CBC mode, then keeping only the last block of ciphertext and discarding the other blocks. The MAC value is then computed by using the last block and the secret key to generate a hash. It is important that the key used in CBC-MAC is different to the key used in CBC encryption. Therefore, during the key exchange process, two secret keys need to be exchange: one for encryption and one for authentication.

The major problem with implementing MACs is that MACs take a long time to compute when compared to computing ciphertext. As the context of the project is to secure instant messaging, it is important to find a MAC that will not excessively slow down the communication exchange, as the messaging will no longer be "instant". The block cipher mode used is CBC, so by using CBC-MAC, the same primitive algorithms are used. This makes proficient implementation easier. A UMAC [5] was also considered, as UMAC algorithms are specifically adapted to particular types of systems to make the generating a MAC multiple times quicker. The main reason why it was not implemented was not used was that finding a Java library to implement UMAC proved to be difficult.

D. Non-repudiation

The aims of non-repudiation are to prevent the denial of previous commitments or actions as well as prove the integrity and origin of the information independently by a third party. By referring back to the generic encryption scenario, it can be seen that non-repudiation involves, like data integrity, a form of authentication.

Bob should ensure that the person he is communicating with is Alice and that her messages are not being altered in any way. Subsequently, at a later date, Alice cannot dispute that it was not with her the communication exchange occurred or that the data Bob received was not what Alice had intended for Bob to receive.

The implementation of non-repudiation therefore involved the same techniques used in entity authentication and message authentication.

E. Key Establishment and Management

To share the secret key used for confidentiality, a key establishment process or protocol must be implemented. This is an important aspect in implementing information security as it deals with how to apply Kerckhoffs' Principle. There are numerous different methods of implementing a key establishment protocol. These methods have a common objective of establishing a shared secret (the key) with an entity whose identity can be verified. This involves is another type of authentication called key authentication, whereby the identity of an entity that can possibly share a key can be verified.

1. Session Keys

The secret key that is shared between two entities is also known as a session key as the key is transient and is only valid for the current communication session. Once the communication session is over, the key is discarded. There were three main reasons why session keys were implemented in the prototype.

The first reason was to remove the need of having to store the secret key. In the case that the entity communicates with a large number of entities, a large number of keys will need to be saved, creating possible storage issues. The second reason was to limit the amount of information given to an attacker. The more ciphertext that is sent with the same key, the more information an attacker has for cryptanalysis. The final reason was to limit the exposure of data if the secret key is compromised. If an attacker discovers the secret key, the exposure of the data (with respect to both time and data quantity) can be limited as the key is discarded upon the termination of the session. Therefore, for the next session, the attacker would have to start again in trying to discover the key.

An advantage with IM conversations is that research has found that most IM conversations last an average of 4.5 minutes [14]. Therefore the attacker does not have long to try to discover the session key before it is discarded.

2. Key Establishment Process

The key establishment process used for the prototype was one where the key establishment was basically a type of message authentication where the message was the secret key. The process involves transporting the secret key over a communication channel using a combination of a public key encryption scheme and a digital signature, shown in Figure 3 - Key Establishment Process.

Alice generates an AES symmetric (secret) key and encrypts the key using Bob's public RSA key. This is sent over the channel where Bob decrypts the message using his private RSA key and takes an MD5 hash of the key. This ensures that the secret key is shared only with Bob. However, Bob still needs to be assured that it was Alice that sent him the key. For that reason, Alice also generates an MD5 hash of the AES secret key, encrypts it with her private RSA key and sends it to Bob. Bob then decrypts the message using Alice's public RSA key. The decrypted message is the hash of the secret key that Alice generated. Bob compares this hash value with the hash value that he generated of the secret key sent in the first message. If the hashes match, then it confirms to Bob that it was Alice that sent him the secret key. There are numerous ways in which Alice and Bob can exchange their RSA public keys, such as sending it via an email. One method implemented in the prototype is discussed in section VIII.B

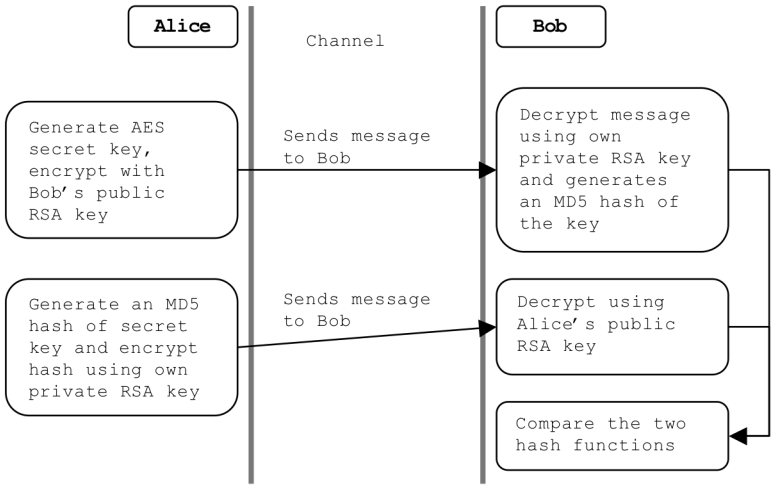

Figure 3 - Key Establishment Process

3. Hash Functions

Hash functions, also known as message digest functions, belong under the unkeyed primitive and are the most versatile cryptographic algorithms. Note that a MAC is a type of hash function, however it uses a key and is thus classed under the symmetric key primitive. Hash functions can be used for authentication, digital signatures and encryption. A hash function operates by taking an input of an arbitrarily long message and produces a fixed-sized result

called a hash. The hash is sometimes called a digest, as it is basically a compact summary of the message.

A hash function generates a hash by mapping the bits of the message string to the bits of a fixed length string. Hash functions are known as many-to-one functions. This is because the message string is longer that the fixed string it is mapped to, so "many" message bits can be mapped to the same fixed string bit. This characteristic implies that collisions (where pairs of different inputs generate identical outputs) are unavoidable with hash functions. In practice, a collision is computationally difficult to find, thus collisions effectively never occurs.

a) MD5

MD5 [5]&[6] is a hash function developed by Ron Rivest, who also designed its predecessor, MD4 [5]. MD5 takes an arbitrarily long message string and generates a 128-bit hash of the message. The message is split into 512-bit blocks, with the last block including the message length and any padding needed to make it 512 bits long.

The blocks are processed in order using a compression function and a 128-bit intermediate state. The state is split into four 32-bit words. The compression function has four rounds in which the message block and the start are mixed using a combination of arithmetic and logical operations. These operations include addition, XOR, AND, OR and rotation operations. After the four rounds, the compression function, the input state and result are added together to construct the output of the compression function.

This type of hash function is known as an iterative hash function. By design, if the compression function is collision resistant, then the hash function is also collision resistant. This has serious implications with the usage of MD5, as the compression function of MD5 is known to have collisions [Appendix C]. In spite of this, there were no known attacks on MD5 itself at the time of implementation. Another problem with MD5 is that its 128-bit hash has a low security level whereby a collision can be found in about 2^{64} evaluations of the hash function.

An alternative hash function that was looked at was the Secure Hash Algorithm (SHA-1) [5][6]. SHA-1 is also based on MD4, but it generates a 160-bit hash. The major problem with SHA-1 is that it too has a low security level, whereby a collision can be found in 2^{80} evaluations. NIST have published a draft standard that outlines three new hash functions based on SHA-1: SHA-256, SHA-384 and SHA-512 [5][6]. The number after "SHA-" indicates the bit size of the hash generated by these functions. These functions are relatively new and have yet to be thoroughly studied, but on the other hand, they provide a higher level of security than MD5 and SHA-1.

The major reason why MD5 was the chosen hash function was that a tradeoff needed to be made in terms of security due to the fact that the hash function was to be implemented in an MSNP-based system. The prototype is restricted in the payload space available. Therefore, the smaller hash size of MD5 made it the viable option in this context.

4. Digital Signatures

A digital signature is designed to uniquely identify an entity, similar to its handwritten counterpart. It is a type of asymmetric (or public) key primitive, whereby each entity has their own key pair consisting of a public key that can be known by anyone (authorised or unauthorised) and a private (secret) key that is known only to that entity.

A digital signature scheme is made up of three algorithms:

- A random key generation algorithm, that generates a key pair
- A signing algorithm that uses a private key to sign a message, creating a signature
- A verifying algorithm that uses a public key to verify a signature

As the signature can be verifiable by a third party without needing the signer's private key, it provides a means for non-repudiation. Digital signatures can also be sued for authentication and data integrity.

a) RSA

The RSA [13] signature scheme is one of the most well known and widely used asymmetric key cryptosystems. Not only able to provide signatures, RSA can also be used for encryption purposes as well. The details of RSA can be found in RSA and [13].

VII. Instant Messaging Design Principles

The prototype requires a communications protocol for setting up and managing encrypted sessions between IM clients. When designing the protocol to be used by the prototype, it was important to define a main objective and then identify key areas that needed to be addressed.

The main objective was to create a protocol that satisfies the security guidelines of section VI while:

- Maintaining compatibility with IM network protocols
- Allowing easy portability between IM networks
- Minimising IM protocol knowledge
- Minimising protocol overhead.

By following these general design rules it was hoped to produce a protocol that provided a high level of security while still remaining practical and compatible with different networks. The actual implementation of the protocol can be found in [Section VIII: Prototype Design].

A. IM Protocol Compatibility and Portability

For the encryption add-on to be compatible with an IM network, the data it generates must within the guidelines of what is allowable on that particular network. It was also important that the protocol be portable between various IM networks. To achieve the best possible compatibility and portability the protocol was designed to take advantage of similarities between the networks.

1. Embedded Plaintext Commands

Prior investigation showed that the popular IM systems exchange messages in plaintext using either UTF8 or ASCII encoding. A command sent as UTF8 encoded text within a message body would therefore be adaptable for use on any network. It was decided that the protocol should operate by exchanging command sequences embedded within the body of a text message.

As using UTF8 encoded commands inside the body of an instant message payload is within the parameters of the IM network protocols, there is less likelihood of the data being rejected by network servers due to 'irregularity'. [figure 4] shows how the encryption protocol is placed within the message payload with an Instant Message.

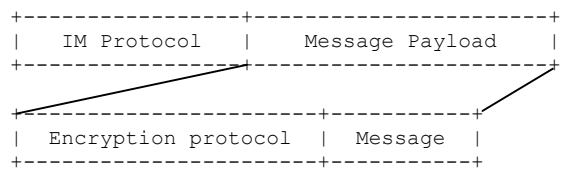

Figure 4 - IM Message Payload

a) Other secure connection methods

As section VI describes, the encryption protocol takes advantage secret-key and public-key encryption algorithms and methods. Many Internet applications that secure communications channels with secret-key and public-key algorithms currently use Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

While TLS and SSL were initially examined as a possible way of securing IM, they were found to be unsuitable for a number of reasons.

TLS/SSL are based on a client-server relationship and are implemented over a special TCP port for the given communications protocol. For example, a secure http session (*https)* would use TLS/SSL over TCP Port 443. There are currently no TCP Port allocations for TLS/SSL sessions using the MSNP and OSCAR protocols.

In addition, with the exception of some ICQ sessions, the public IM networks do not establish direct connections between users when message exchanges occur, meaning a direct TLS/SSL connection between two subscribers would not be possible in most circumstances. Establishing peer-to-peer connections or using non-standard ports would contravene the design objective of working within existing IM protocol parameters.

2. Reduced IM Protocol Knowledge

The encryption protocol does not rely on information generated by or unique to a specific IM protocol. By minimising the amount of information required for the protocol to function, it reduces the amount of changes needed for use on different networks. Therefore, the protocol deals only with text strings, and contains no knowledge of the system in which it is being used.

B. Limiting Additional Information

The amount of information that can be carried within a message payload is limited by the IM protocols [25]. As the process of encrypting a plaintext message expands the length of the message, it was important to try and minimise the amount of additional protocol data sent with the ciphertext.

Methods used to reduce the amount of information exchanged included using short command codes and not transmitting Initialisation Vectors (VIII).

VIII. PROTOTYPE DESIGN

A. Message Capture and Encryption Protocol

The encryption prototype operates by intercepting each message typed by the user before it is transmitted to the IM server. It then scans the intercepted text for a special escape sequence and command code, which determines whether any operation should be performed on the text. The message is then passed back to the IM client for transmission.

When receiving messages, this process is reversed. Messages are captured by the prototype as they are received and are scanned for command codes before being passed back to the client for display.

The encryption software operates without knowledge of the IM network or client being used, dealing only with text. This approach allows the software to be used with any plaintext based messaging system with little or no modification.

Modifications to the TjMSN client consisted of message interception code and minor changes to the Graphical User Interface.

1. Escape Sequence

A special escape sequence was used to identify messages that required some form of action. The sequence "###@ " (including space) was chosen, as it is unlikely to proceed any message during a normal conversation.

As text is passed to the add-on, any messages beginning with this sequence are further examined for a 'command code'. Messages that do not contain the escape sequence are passed back to the client unaltered and are transmitted to the recipient.

2. Command Code

A two digit numerical code allows the add-on to communicate and determines what actions should be performed on the plaintext message. A two digit numerical code was chosen rather than a textual code due to the limited space available in each message payload.

The structure of the code is similar to the implementation used in the OSCAR protocol [25], with each code divided into a 'family' and 'instruction'. A code represented by the value *XY*, for instance, would indicate *Family X; Instruction Y*. When a command code is detected, the appropriate operation is completed and if necessary a new command code will be substituted in its place.

a) Code Families

The current protocol consists of four code families, with each family defined by the action to be taken. Table 1 - Command Code Families summarises the command codes that are currently defined.

Family	Code	Action performed
1	**1**	Send request for Encrypted Session
	2	Received request and encrypted key
	3	Create and send signature
	4	Received signature for verification
	5	Send acceptance of Request
	6	Received acceptance of request
2	**1**	Message to be Encrypted and transmitted
	2	Received encrypted message
	3	Part 1 of split message*
	4	Part *n* of split message*
	5	Final part of split message*
3	**0**	Start encryption end sequence
	1	Received notification of end sequence
	2	De-activate encryption and send final end code
	3	Received end code, de-activate encryption
4	**1**	Generate and send certificate
	2	Received Certificate

Table 1 - Command Code Families

*Commands defined but not currently required in prototype

Command codes can be categorised as being either *internal* commands or *external* commands. Internal commands provide instructions to the local client and can be generated by buttons on the chat interface, while external commands are generated from within the encryption prototype and are transmitted to the remote client. An example of an internal command is the code (11), which, when the 'activate encryption' button is pressed, tells the add-on to generate a new AES key and transmit this to the remote user with the external code (12).

3. Secret Key Management and Use

Each encrypted session uses a shared 256-bit AES key which is randomly generated and disposed of after the session has been completed. A 128-bit AES key also generated for use with the AES-CBC Message Authentication Code (MAC).

a) AES Key Generation

The 256-bit and 128-bit AES keys are generated on the computer of the individual who initially requests an encrypted session. The source of randomness used to create the key is Sun's Pseudo Random Number Generator (PRNG) [16]. The 256-bit key is used to encrypt and decrypt messages, and also to generate nonce Initialization Vectors for the encryption and decryption process. The 128-bit key is used to create a MAC of the ciphertext generated by the 256-bit key for each message.

b) Key Exchange and Authentication

Once session and MAC keys are generated, they are encrypted using the RSA public key of the recipient of the encryption request. An MD5 hash of the combined keys is also encrypted using the private key of the initiator, producing a digital signature. The encrypted keys and signature are then transmitted to the recipient.

The recipient decrypts the secret keys using their private key, and the MD5 hash using the initiators public key. An MD5 hash of the decoded keys is then produced to compare with the hash given in the signature. If the hashes match, the keys are saved and the recipient may accept the request and an encrypted session may begin. If the hashes do not match then it can be presumed that the data has been altered in some way in transit, or incorrect keys were used at some stage of the process.

c) Synchronised Message Counters

When in an encrypted session, a counter kept by both users is incremented every time a message is sent or received, starting from zero. This allows each message to be uniquely identified by the state of the counter when it was sent or received. The importance of the synchronised counter is explained in the sections *Message Encryption* and *Nonce Initialization Vectors*.

d) Message Encryption

Messages are encrypted using a 256-bit AES key initialised in CBC mode with PKCS#7 Padding. The encrypted blocks are cycled using a random Initialisation Vector, its source being derived from the message counters. A Message Authentication Code is added to the ciphertext message to ensure that the ciphertext is not altered in transit.

e) Nonce-generated IV

An integer message counter is used with the secret key to create a nonce-generated IV for each message that is to be encrypted. Starting from *0*, the message counter is synchronised between the users and increments by *1* for each message. The state of the counter *n* is encrypted using the secret session key *k* for every message, producing value *k(n)*, which is used as a source of randomness to initialize the AES cipher before encrypting or decrypting a message. The value of *k(n)* used to encrypt and decrypt a message can never be re-used. It can be considered random as *k* is unique to every session, while any instance of *n* can be used only once per session.

An advantage of using the synchronised message counters to create the IV is that the IV need not be transmitted with the encrypted message. This prevents an attacker from replaying any messages to the users, while it also prevents an attacker from blocking messages, as this would cause the counters to become out-of-sync. Not transmitting the IV also helps to reduce the protocol information in the message payload.

A drawback of the nonce-generated IV system is that due to the process of converting an integer counter to a type suitable for generating the IV, a limit of 10,000 messages is applied to each session. This can be seen as an acceptable limitation, as the number of messages sent using a single key should be limited for security reasons.

f) Message Authentication Code

Before the ciphertext is transmitted, a MAC is created. The prototype uses the AES *CBCBlockCipherMAC* method provided by the Bouncy Castle package. The 128-bit secret key is used for this function, which produces a 32-bit sequence. This sequence is added to the front of the ciphertext message before transmission. When a message arrives, the received MAC is compared with a newly generated MAC of the received ciphertext. Matching MACs will allow the message to be deciphered.

4. RSA Key Generation and Management

The Java KeyTool utility was used to generate and store the RSA public/private 'key pairs' used when developing the software, although other methods are available. This process is shown in KeyTool. X.509 Certificates (public keys) exported from a KeyStore can be transported in a number of ways, such as email or on diskette or flash drive. The add-on also contains the ability to export and import X.509 public keys from within the IM client.

B. Key Generation, Exchange and Authentication

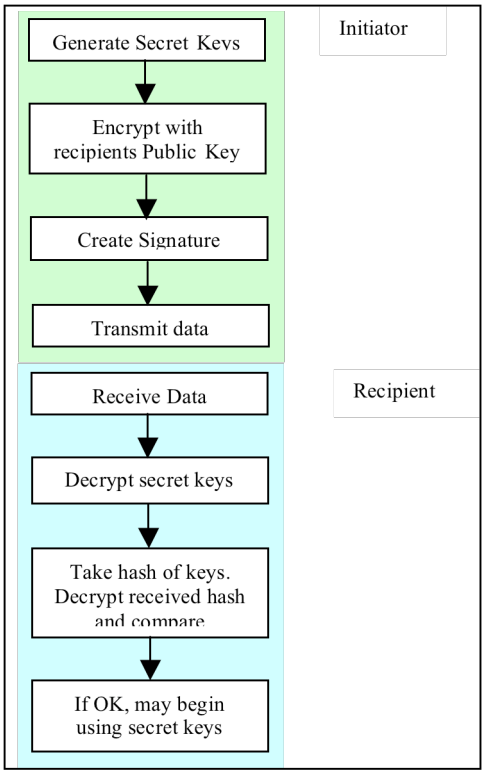

Figure 5 - Key Generation and Exchange

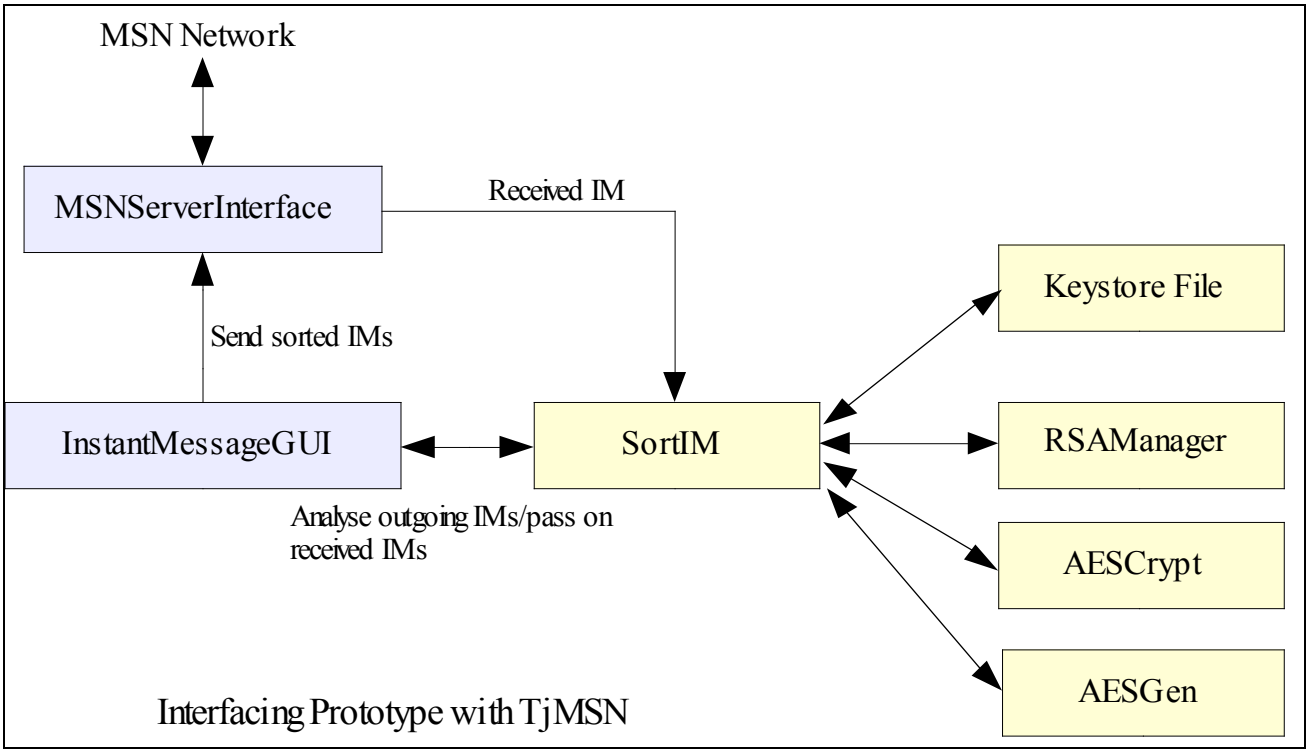

Figure 6 – Interfacing Prototype with TjMSN

C. *Protocol Processes in Detail*

1. Interfacing Prototype with TjMSN

Figure 6 – Interfacing Prototype with TjMSN above shows the main interfaces and classes used to provide encryption. TjMSN classes are highlighted blue while the prototype classes are yellow. The code of these classes can be found in the appendix.

SortIM provides the main link between TjMSN and the encryption code. When a new chat session is established, TjMSN executes the InstantMessageGUI class, which provides an area for sending and viewing messages (Figure 7 - TjMSN Conversation Window). Additional code written into InstantMessageGUI initialises SortIM at the same time, and begins diverting messages sent or received.

When a message is to be sent, SortIM first analyses it for a command code, before passing it back to InstantMessageGUI which forwards it to MSNInterface. Received messages are passed straight to SortIM before being passed to InstantMessageGUI for display. This occurs whether or not encryption is active. Information is passed between the classes as plaintext strings.

Although SortIM is responsible for managing the encrypted sessions, it does not have any knowledge of the IM network protocols.

The classes RSAManager, AESCrypt and AESGen are accessed as needed by sortIM, as is the user's keystore file. The encryption classes have no knowledge of either the MSN Protocol or the encryption protocol and will simply perform a function on a string of text.

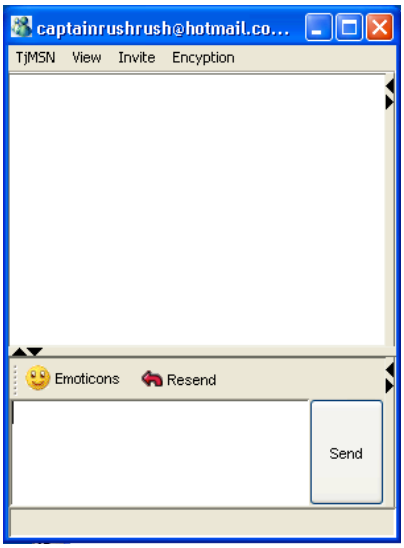

Figure 7 - TjMSN Conversation Window

2. Protocol Functions

This section details the sequence of command codes when performing the main functions of the prototype.

a) Activate Encryption

When a user requests and encrypted session, the internal command codes "`###@ 11 `" and "`###@ 13 `" are generated and sent as two successive instant messages. These messages are detected by the SortIM before being transmitted and are replaced with two new messages, the

first being "`###@ 12 +encrypted keys`" and the second being "`###@ 14 +Signature`". That is, the initial code (11) causes the software to generate the two AES keys, encrypt these with the public key of the recipient and transmit with the command code (12). The second code (13) causes the software to transmit the digital signature associated with the encrypted keys with the command code (14).

Upon receiving the first code (12), the keys are decrypted and saved. When receiving second code (14) the signature is verified and, if valid, the user is shown a dialog box allowing them to accept or decline the request. If accepted, the internal code (15) is generated to activate encryption using the saved key and to transmit the request accept code (16). When the requester receives the accept code (16), encryption is activated.

The diagram below shows the internal and transmitted codes during this sequence.

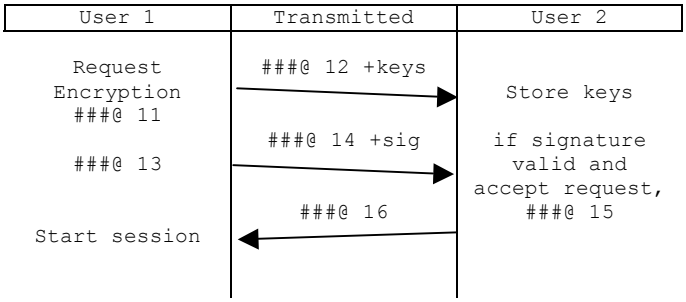

Figure 8 - Activate Encryption

b) Encrypted Message Exchange

During an encrypted session the internal command code "`###@ 21` " is added to the front of each message to be sent. This informs the add-on that the message should be encrypted. The text "`###@ 22 +message`"is encrypted, and a MAC is produced from this ciphertext.

The receiver first checks the MAC and, if valid, will decode the ciphertext. The command code (22) is then recognised and the message text is passed to TjMSN for display.

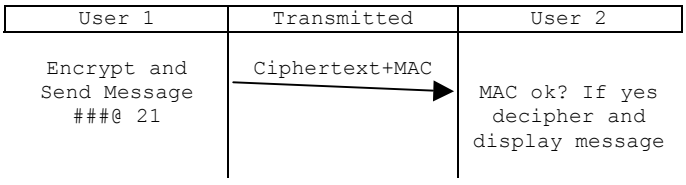

Figure 9 - Encrypted Message Exchange

c) End Encrypted Session

When either user decides to end the encrypted session, the internal code (30) is generated. This in turn causes the message "`###@ 31` " to be encrypted and transmitted. Upon receiving the code (31), the internal code (32) is generated. The (32) code de-activates the encryption and replies with the code (33), used to confirm that encryption has been de-activated. When the initiator receives the (33) command, encryption is turned off and a confirmation message is sent in plaintext.

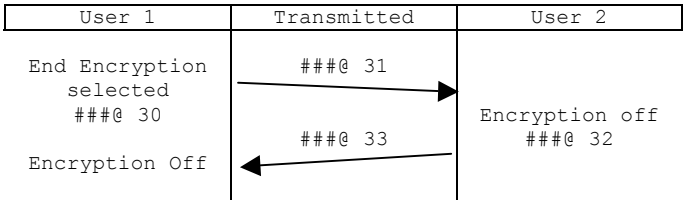

Figure 10 - End Encrypted Session

Encryption can also be ended simply by closing the messaging window (all keys and configurations are discarded).

d) Export Public Key Certificate

When "export public key" is selected by a user, the internal code (41) is generated. This code causes the public key associated with the configured private key alias to be retrieved and converted into an X.509 certificate encoded in base64. This is then transmitted as "`###@ 42 +certificate`". Upon being received the certificate can be named and is stored in the directory of TjMSN.

IX. Prototype Testing

As the prototype matured testing switched from 'test harness' console programs to testing on the MSN Network. This process involved observing the behaviour of the prototype while undertaking regular IM conversations. The section *Software Demonstration* shows the use of the prototype under normal circumstances. The *Performance Considerations* section contains analysis of the time delay introduced in the process of encrypting and decrypting messages on a number of systems.

A. Software Demonstration

1. Preliminary information

After being compiled, the modified TjMSN client used for demonstrating the prototype is packaged into a single Java Archive (JAR) executable. The file can be run from either the command line using '`java -jar filename.jar`' or by double clicking the icon from within an operating system GUI. Java KeyStore files are placed in the same directory as the JAR file.

A simple demonstration of was undertaken to show the setup of an encrypted session and a small exchange of encrypted information. The test was performed using two Windows XP machines on the same LAN connected to the MSN Messenger network.

Data traffic generated by the clients was captured using the 'Ethereal' packet sniffing software. The filter string [21]"`tcp port 1863`" was used so that only MSNP related data was captured, allowing for easier inspection.

Excerpts of captured data are shown in ASCII format rather than hexadecimal, as MSNP is an ASCII based protocol [25]. As such, commands such as 'new line' (`\n`) and 'carriage return' (`\r`) have also been omitted. Information not directly relating to the encrypted exchange, such as TCP/IP/Ethernet header fields have been omitted. The captured information is shown from the perspective of the principle who is receiving the request for an encrypted session.

2. Demonstration

Initially a chat session was established by launching the Instant Messaging GUI. A menu has been added to this window allowing control over the various encryption options. This menu is shown in Figure 11 - Encryption Menu.

Figure 11 - Encryption Menu

At this stage, any instant messages are exchanged in plaintext. The following excerpt shows how the message "hello" appears when received:

```
MSG captainrushrush@hotmail.com nigel 69
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

hello
```

Before an encrypted conversation can take place, both users must configure the Java KeyStore that they are using, using the 'Certificate Setup' option from the menu. The dialog box for Certificate Setup is shown in Figure 12 - Enter KeyStore Information Dialogue.

Figure 12 - Enter KeyStore Information Dialogue

Filename: The name of the KeyStore file
Password: The password used to access the KeyStore
Private Key Alias: The name assigned to the Private Key to be used during encryption.

The KeyStore file is placed in the same directory as the TjMSN JAR file. The details supplied are tested for validity after "OK" is pressed.

Once both parties have configured the KeyStore, the 'Activate Encryption' option can be used to initiate key exchange. Selecting this option opens the 'Alias Configuration' dialog shown in Figure 13 - Enter KeyStore Alias Dialogue.

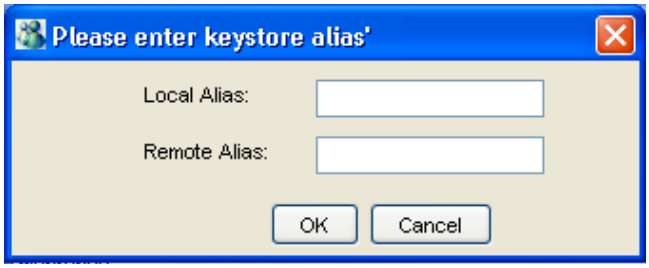

Figure 13 - Enter KeyStore Alias Dialogue

Local Alias: The alias for the private key stored locally (a.k.a. Private Key Alias).
Remote Alias: The public key alias of the recipient of the request. This key is used to encrypt the secret key.

Once "OK" is pressed, 256-bit and 128-bit AES keys are generated, encrypted using the public key of the recipient, and transmitted. An MD5 hash of the key is also made, which is encrypted using the private key of the requester and is sent immediately after the secret keys. The encrypted keys, as received, are shown in the following excerpt.

```
MSG captainrushrush@hotmail.com nigel 246
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

###@ 12 OA5rd8xeF/pt85Kqko4zeKrf0XeyM9acYYKOcFj
ZO9XVOsSKOcVzqCy7FyNeL9pB8qfBtIqQT6KeO4tgqpd+2Y
aJLoyzpOgTGlHZuS1xJJOMZSRkqk3G+9as9b5RAr1Pkvphf
Ped9R+SZaQ7oA6oadd1akhBO8HPsRuX9iMJ5dg=
```

It is important to note the MIME header of the incoming message. The "`content-Type: text/plain`" signifies that this information has been transmitted as a standard message. The first line simply indicates the MS Passport of the sender and their 'nickname'.

The code at the beginning of the payload is recognised by the software as being encrypted AES keys. The string is decrypted and the keys are saved as key objects to be used if the request is accepted. No other action is taken at this point.

The following message immediately follows the first.

```
MSG captainrushrush@hotmail.com nigel 246
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

###@ 14 5C5iXd/NHGCDQefBrnZ8eSx9qXQcC7AvEX+or9v
WqUdQ0JXRro/fzzsoT2Ynr+OfD3nAhfznAspjqf+OMbN/qI
FwqPxuMFDEp00icQnKQ9KFuGSsjFdK3TOmbaESyH1LoeGOK
CECnjmyT5y20jIiHY8SulKFaYNtVVJgxruT8VU=
```

It contains an MD5 hash of the keys encrypted with the private key of the sender. The hash is decoded using the public key of the sender and compared with an MD5 hash of the previously decoded keys. If the hashes do not match, a message is sent notifying the sender of this and the keys are discarded. A successful hash will cause the dialogue box shown in Figure 14 - Enable Encryption Session Dialogue to appear.

Figure 14 - Enable Encryption Session Dialogue

Answering 'no' or 'cancel' results in the key being discarded and the requester being notified. Answering yes results in the following message being transmitted:

```
MSG 3 N 70
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

###@ 16
```

This message, when received by the requester, will inform them that the request has been accepted and will place their client into encrypted mode. All communications between the clients is now encrypted using the secret key.

For demonstration purposes, the message "`hello, how are you?`" was sent. The following shows the data as transmitted over the Internet.

```
MSG captainrushrush@hotmail.com nigel 106
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

Ns8K/4==5B5+RU3c/KNaDkuWr0/SWskVj78QHw6FniDjt6G
v3+U=
```

The (MAC) is first removed and a new MAC is generated using the 128-bit key and the ciphertext. If the MAC matches, the string is decrypted and the original message is passed to the client. Replying to this message with "good" produced the following output:

```
MSG 8 N 86
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

mt57KN==awNph3Yr3YO2EVcdGSreYg==
```

At this stage, any protocol control information is also encrypted to prevent a third party from tampering with the conversation by sending dummy commands.

To end an encrypted session, a user simply selects the option from the menu. This sends a notification to the other user and upon receiving acknowledgment the encryption is de-activated. The users are notified in the message history window that the encryption has been de-activated. The following sequence shows the initial end encryption command as received.

```
MSG captainrushrush@hotmail.com nigel 94
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

Rg85aQ==Dbgyvh8q5yC0hPq/tnci0Q==
```

Encryption is then de-activated and the following message is automatically sent in reply:

```
MSG 6 N 70
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8

###@ 33
```

This message informs the other user that encryption has been de-activated. The image below shows the messages seen in the history window from the perspective of the initiator.

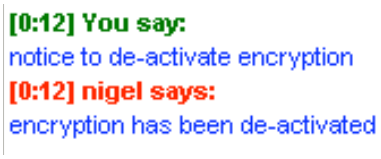

Figure 15 - Encryption Deactivation Message

When receiving a public key exported by another user, the following dialog box appears:

Figure 16 - Certificate Alias Dialogue

This is the name of the file in which the received certificate is stored. Pressing cancel will discard the received certificate. When 'OK' is pressed, the following information appears in the received message window:

Figure 17 - Certificate Storage Confirmation

Although this certificate can now be imported and used to decrypt information, it is highly recommended that the users confirm the authenticity of the certificate by verifying the certificate fingerprint over some out-of-band channel, such as by telephone.

B. Performance considerations- Benchmarks

Encryption can be a mathematically intensive process and thus a certain amount of CPU overhead is incurred as a result of encrypting or decrypting a message. This overhead is noticed as a time delay as the plaintext is converted to cipher text.

Due to the near-realtime nature of instant messages, it was important to observe the delay introduced by an encryption process of the specifications used in the add-on. By recording the time-delay experienced on a number of different systems it is possible to predict the approximate performance on a given configuration. This in turn can indicate the baseline configuration for tolerable performance.

1. Benchmark Design

To simplify the testing process it was decided that the benchmarks should consist of standalone Java console applications. This allowed for easy installation and testing on various machines and did not require an active internet connection.

The java classes used to perform the encryption in the benchmarks were the same classes used in the final implementation of the software. This was done to ensure that the results provided would reflect the time delay experienced using the actual software. Special 'test harness' classes were written to access the encryption functions, provide 'dummy' information and to perform timing.

2. Limitations due to Clock Time Granularity

The method `System.currentTimeMillis();` located in `java.lang.System` was used to time the length of the various operations. As its name suggests, this method returns the time of the system accurate to three decimal places. The time elapsed was recoded by calculating the difference between the time obtained before and after an operation was called.

A limitation of this method is that the accuracy of the time obtained is dependant on the *Time Granularity* [22] of the system clock. Time granularity can be defined as the time interval that elapses between successive updates of the system clock. It may also be described as the effective resolution of the system clock. This interval varies between operating systems and different configurations of the same operating system. Time granularity causes timing inaccuracies for very short periods of time System Clock Granularity.

Therefore, to increase the accuracy of the measurements, it was necessary to extend the length of the operations being timed. This was done by placing the

operation in a loop and dividing the resulting time by the number of loops performed.

The results obtained provide an approximate guideline to of prototype performance.

3. Test Machines

The benchmarks were run on a number of different systems that might typically be found in an office and also on a significantly outdated system for comparison. This included both desktop and portable systems. All machines were tested using the J2SDK 1.4.2 with the Bouncy Castle Provider and 'unlimited strength' US Export Policy installed.

System A	AMD Athlon XP2400+ 512MB RAM Windows XP Pro SP1
System B	Apple iBook G4 800Mhz 384MB RAM Apple OS 10.3.5
System C	Intel Pentium M 1.4Ghz 256MB RAM Windows XP Home SP1
System D	Intel Celeron 400Mhz 256MB SDRAM Windows 2000 Pro SP1*
System E	Intel Pentium 4 2.66Ghz 512MB RAM Windows 2000 Pro SP3

Table 2 - System Properties

The benchmark was run on each machine three times, with the average of these three results being shown in the graphs which follow. The complete table of results can be found in System Clock Granularity.

4. AES Benchmark

The AES benchmark tested the three symmetric key functions used by the add-on, those being key generation and encryption/decryption in CBC mode. Early trials found the encryption and decryption process taking less than 16ms, so a loop was used to increase the time interval between polling the system clock. The number of loops used was 50,000.

Two different length strings were used in the benchmark, simulating a short sentence (58 bytes) and a longer sentence (108 bytes) that would be typical of an IM conversation. An example of the program output is shown in [Figure 18]. Although capable of generating an AES key, the benchmark would not run correctly on the MAC OSX system, thus AES results for the iBook have been omitted.

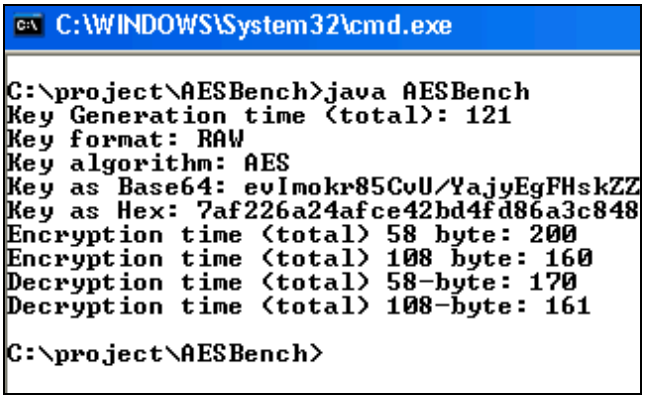

Figure 18 - AES Testbench Example

a) Key Generation

The following graph Figure 18 - AES Testbench Example shows the average time taken for each system to generate the AES key. This operation was not looped.

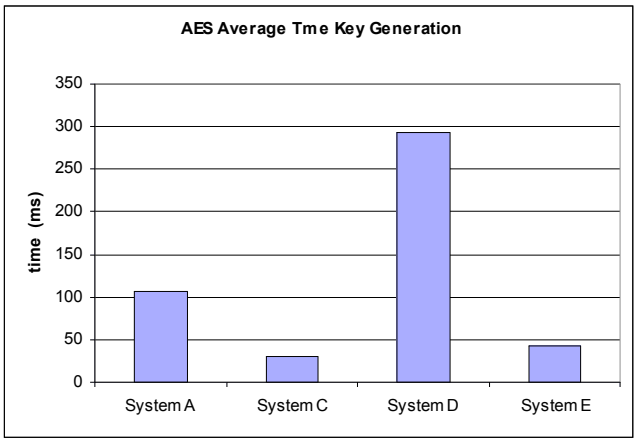

Figure 19 - Average Time for AES Key Generation

Intel processors have a notable speed advantage in generating AES keys, as these results show. Although System D did not show a high average performance, the complete results tables show that on two of three occasions it was able to generate the key at roughly the same speed as System A.

b) AES Encryption Functions

The following graphs show the average of the results obtained from the AES benchmark. Each operation was looped 50,000 times.

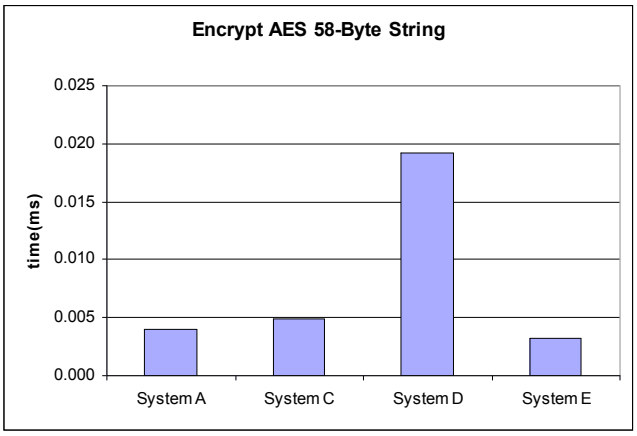

Figure 20 - AES Encryption with 58-byte String

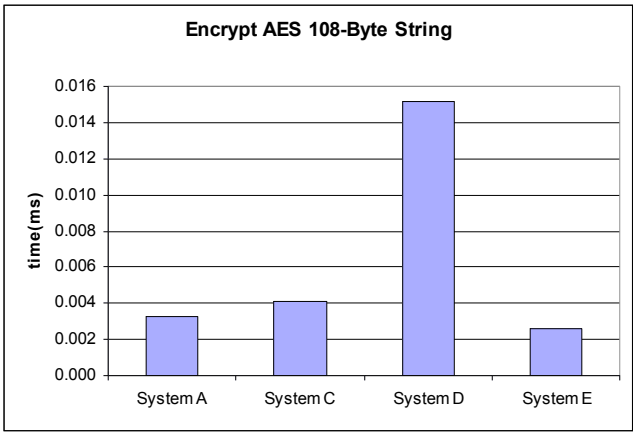

Figure 21 - AES Encryption with 108-byte String

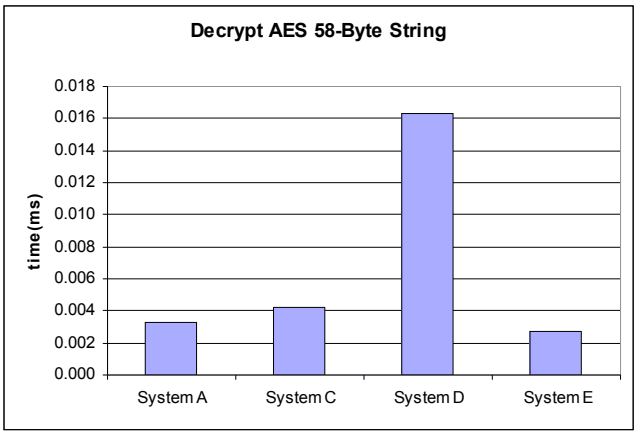

Figure 22 - AES Decryption with 58-byte String

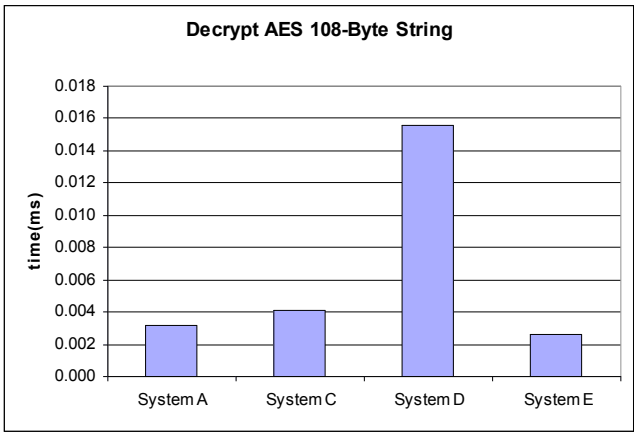

Figure 23 - AES Decryption with 108-byte String

The results show that the AES encryption functions are quite fast, even on the slowest machine used. For example, System D took between 0.015ms and 0.020ms for encrypting the 108 and 58 byte strings. The more modern processors were able to stay below 0.006ms for all of the operations. These times suggest that it is extremely unlikely that users would notice any additional delay in the exchange of messages due to the AES operations.

A conversation between two clients using a machine of the specifications of System A might expect an additional 0.010ms of latency, a near instantaneous period of time.

5. RSA Benchmark

The RSA benchmark first generated an AES key (as the message) before timing the following operations:

- Signing a message
- Encrypting message with a public key
- Decrypting message with private key
- Verifying a signature
- Encoding with private key
- Decrypting message with public key.

As asymmetric encryption is more processor intensive than symmetric, the operations were looped 500 times. Figure 24 - RSA Testbench Example shows the output of the RSA benchmark program.

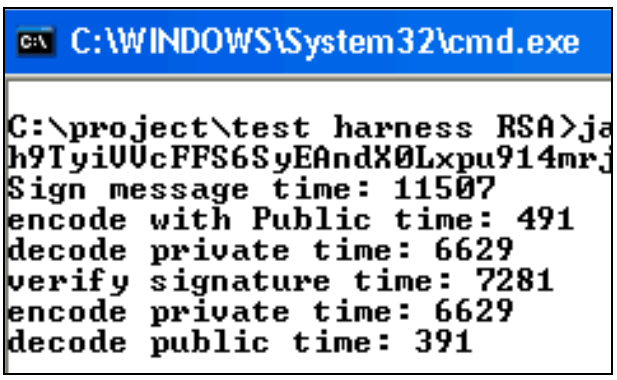

Figure 24 - RSA Testbench Example

The following graphs show the result averages for the RSA benchmark program. Although 'encoding with private key' was included in the benchmark, a graph of the results is not shown as this function is not specifically used on its own by the prototype (it is used in combination with an MD5 hash). The results are included in table form in Benchmark Tables of Results.

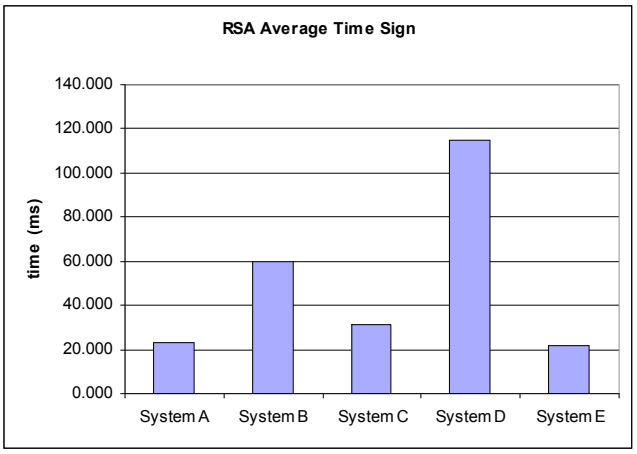

Figure 25 - Average Time for RSA Signing

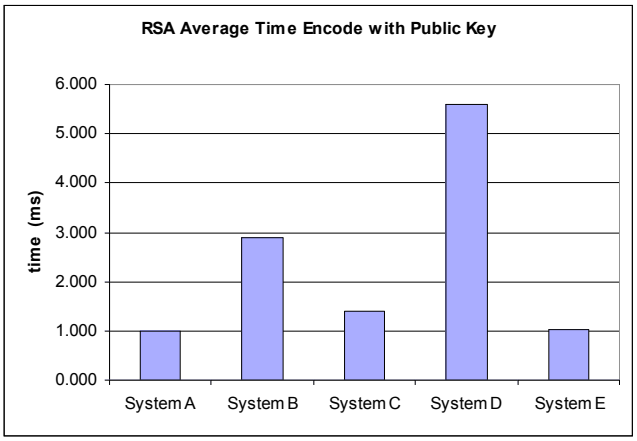

Figure 26 - Average Time for RSA Encryption

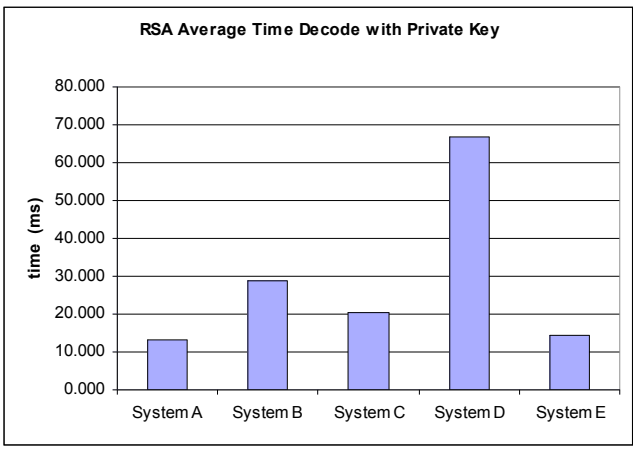

Figure 27 - Average Time for RSA Decryption (Private Key)

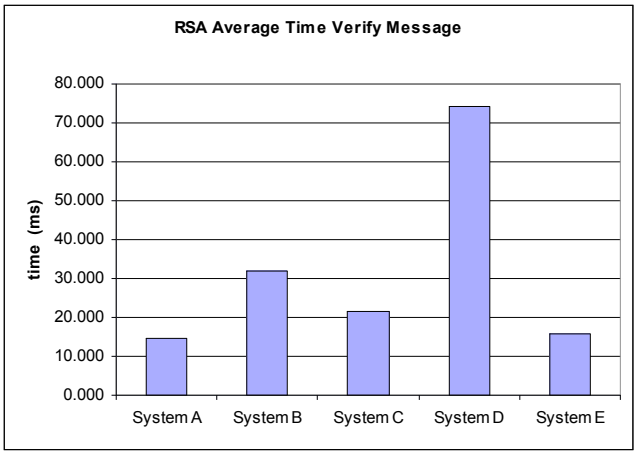

Figure 28 - Average Time for RSA Message Verification

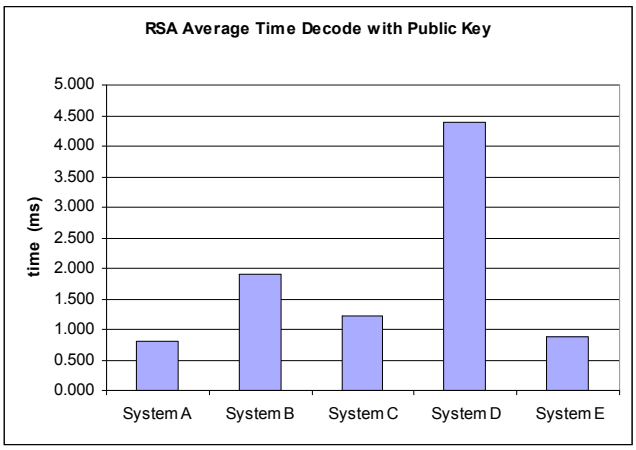

Figure 29 - Average Time for RSA Decryption (Public Key)

RSA functions took considerably longer than AES, as expected. The most time consuming tasks were those related to authentication, as signing and verifying messages are both two-part processes. However, the results for the signing and verifying benchmarks suggest that the main cause of delay is not from the RSA algorithm itself but through creating the MD5 hash.

For example, verifying a message requires that the received ciphertext is decoded using the public key of the sender, after which an MD5 hash is taken of the message for comparison. The 'decode with public' graph shows the decoding process to be relatively quick, with a time of 4.5ms for the slowest system. This would suggest that, on the same system, a signature given for verification might take roughly 5ms to decode and 70ms to create and compare MD5 hashes.

'Encode with private' also supports this, as measured private key encryption times occupy only a small percentage of the total time taken to sign a message.

By adding the results in terms of sequence-of-occurrence it is possible to estimate the approximate delay expected when using the RSA functions. For example, the processing time required to initiate an encrypted session between two machines of System D specifications would be (including key generation time for the two AES keys):

```
Key generation: 586ms
Encode AES key with public: 5.6ms
Sign message: 114ms
Total: 412.6ms
Decode AES key with private: 4.3ms
Verify signature: 74.4ms
Total: 78.7ms
```

In this scenario, the time elapsed between one user requesting an encrypted session and the other having the option to accept or reject the offer is approximately 883ms in addition to any network latency. While this may seem like a long delay, the figure is approximately 271ms in addition to network latency when using a relatively recent system such as System A. The fast AES key generation ability of recent Intel processors further reduces this time delay.

It should be noted that the RSA functions are used only at the beginning of an encrypted session, after which the significantly faster AES encryption is used. In actual use, the RSA-caused delay was found to be quite acceptable.

X. DISCUSSION

A number of design constraints had to be considered when trying to achieve a balance between information security objectives and a practical solution. As IM is a near real-time exchange of short messages, speed and message size were critical issues.

When choosing cryptographic algorithms, an important assessing factor become the speed at which the algorithm could operate for the required level of security. A slow algorithm would introduce a noticeable latency to the conversation, while a fast but insecure algorithm would not meet security requirements.

The most crucial factor, it turns out, was not the encryption algorithms themselves but the speed and size of the hashes used to authenticate the encrypted messages, as shown in the performance testing. This compromise on hashes can be seen in the use of CBC-MAC and MD5 in the place of algorithms that are more secure, but unsuitable for IM systems.

CBC-MAC, which gives 64-bit security, was used in favour of the slower and larger HMAC, while MD5 provides a 64-bit hash at a greater speed than a more desirable but larger hash such as SHA-256. While these hashes are not the strongest solutions cryptographically, they provide a practical balance between high security and reduced size and delay.

Additionally, the cryptographic world is constantly evolving and new algorithms became available later in the design that would have been useful during the development process. An alternative to using RSA was to use the Elliptic Curve Cryptography (ECC) system [28]. ECC has the advantage of having what is known as a 'strong key'. This means that a shorter ECC key would provide the same security as a longer RSA key. For example, a 160-bit ECC key would provide the same level of security as a 1024-bit RSA key.

An algorithm with such small keys would significantly reduce the amount of data transmitted during secret key exchange, possibly reducing the number of steps required. The code libraries used in the project did not include ECC until late in the development stage when Java released JDK5.0.

An issue that emerged later in the development stage was that of user interface and user interaction. Originally not given the same weighting of importance as technical issues, it became apparent that the user interface and how the user interacted with the prototype functions was also a significant factor in protocol design.

Overall, when designing such a cryptographic system, the importance of planning becomes apparent. Such systems are only as strong as the weakest link, and the group found that careful planning made work easier and produced a more secure system.

XI. RECOMMENDATIONS

The prototype was designed as an interim solution while managers developed an IM usage policy or implemented enterprise IMs. IM usage in the workplace has greatly increased in prominence over the last several years, and as such will likely receive more attention from management in terms of policy and the integration of enterprise IM solutions. Therefore, the lifespan of the prototype is expected to be short.

This means that although there is room for improvements can be made with the prototype, some elements are sufficient to last over the expected lifespan of the application.

For example, the lifespan of AES is expected to be about 10-20 years [6]. AES was standardised at the end of 2001, so the lifespan of AES is expected to surpass that of the prototype. Therefore, at this point in time, there is no need to consider replacing AES.

There are, however, a number of improvements that can be made in the short term. The tradeoffs mentioned in the Discussion section need to be further evaluated, taking into account recent changes in cryptographic code libraries. For example, replacing RSA with ECC might be a consideration as it reduces the key length used.

The current use of MD5 and CBC-MAC, while sufficient, is not ideal. A more comprehensive analysis of hash and MAC algorithm combinations to find an optimal balance between security, speed and size would be part of further development on the prototype.

A more exhaustive test of the Sun PRNG would also be undertaken. Finding a good source of randomness is crucial when generating the secret keys used for message encryption. Further tests on the PRNG would provide an indication of its long-term viability.

The method by which the public keys are exported could be made more secure by adding authentication functionality. Currently, when Alice and Bob exchange public keys, it is done so over an insecure channel. Although public keys can be known by anyone, Eve can still cause difficulties by intercepting the public key and replacing it with one that Eve had generated.

A GUI improvement would be to provide an indication to the user as to whether they are currently in an encrypted conversation or not. This was attempted during the development but the group was unable to integrate such a function consistently as there are several bugs in the TjMSN code.

Lastly, simplified key generation and management, perhaps through the inclusion of a dedicated GUI tool, would provide an easy way for users to manage their asymmetric keys, as the current console-based method is not user-friendly.

XII. CONCLUSION

Instant messaging has expanded greatly from its humble beginnings as an Internet gossiping application. IM applications are now widely used in the workplace for many purposes such as organising impromptu meeting and obtaining real-time answers to work related queries.

A major problem with workplace IM usage is that insecure public IM networks are more widely used than more secure enterprise IM systems. In addition, managers have yet to be proactive in developing an acceptable IM usage policy or considering enterprise IM.

The concept behind the prototype was that it would be used as a short-term solution to the current security problems that public IM usage in the workplace posed. Additionally, the context of use that was focused on was for small companies that did not have the monetary funds or technical resources to consider alternative solutions, or companies consisting of several people communicating over the Internet. Several current solutions to the public IM problem were identified, but must be purchased and thus do not provide for the context that was focused on.

The solution that was decided upon was to design and develop an IM add-on that implemented information security objectives that would not adversely affect any IM functionalities. To do this, some tradeoffs were made in terms of level of security and IM operational characteristics. These solutions still provided the level of security required, though.

Testing found that the prototype was able to function well in the manner designed, and performed all functions as expected. The speed at which encrypted messages were exchanged was in fact faster than expected and the prototype did not negatively affect the real-time characteristic of IM applications. Recent cryptographic code library updates to include algorithms such as ECC also provide new avenues of refining and improving the prototype.

Overall, the prototype was able to demonstrate the plausibility of encryption over a public IM network, and with additional testing and development, could provide simple and free cryptographic solution.

XIII. REFERENCES

[1] Blue Coat Systems, Inc, "Establishing an Internet Use Policy for Instant Messaging" Whitepaper, http://www.bluecoat.com, 2004 (as of June 3rd 2004)

[2] Websense, "Key Internet Usage Statistics", http://www.websense.com/company/news/stats.php

[3] H. de Vos, H. ter Hofte, H. de Poot, "IM [@Work] Adoption of Instant Messaging in a Knowledge Worker Organisation", Telematica Instituut, 2004 (as of June 3rd 2004)

[4] National Institute of Standards and Technology, "Special Publication 800-57: Recommendation for Key Management", June 2003

[5] N. Ferguson, B. Schneier, "Practical Cryptography", Wiley Publishing Inc., 2003

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

[7] BBC h2g2, "Basic Cryptanalysis", http://www.bbc.co.uk/dna/h2g2/alabaster/A613135 (as of June 3rd 2004)

[8] National Institute of Standards and Technology, "FIPS-PUB-147: Advanced Encryption Standard (AES)", November 2001

[9] National Institute of Standards and Technology, "FIPS-PUB-196: Entity Authentication using Public Key Cryptography", February 1997

[10] A. Berent, "Advanced Encryption Standard by Example", http://www.abisoft.net/documents/AESbyExample.htm, (as of November 21st 2004)

[11] N. Furguson, "The AES Block Cipher", http://th.informatik.uni-mannheim.de/people/lucks/papers/Ferguson/AEScipher.ppt, (as of November 21st 2004)

[12] L.Dobrow, "Study: IM Soaring Among Adults, But At-Work Usage Less Than Expected", www.mediapost.com/dtls_dsp_news.cfm?newsID=267165, Thursday, Sep 02, 2004 (as of November 21st 2004)

[13] RSA Laboratories, "Frequently Asked Questions About Today's Cryptography, Version 4.1", http://www.rsasecurity.com/rsalabs/node.asp?id=2152, 2004 (as of November 21st 2004)

[14] E. Isaacs, "A Closer Look at Our Common Wisdom", ACM Queue 1(8), November 2003

[15] The Legion of the Bouncy Castle, http://www.bouncycastle.org, (as of November 21st 2004)

[16] Class Secure Random, http://java.sun.com/j2se/1.4.2/docs/api/java/security/SecureRandom.html, (as of November 21st 2004)

[17] Class CBCBlockCipherMac, http://www.bouncycastle.org/docs/docs1.4/index.html, (as of November 21st 2004)

[18] Ethereal, www.ethereal.com, (as of November 21st 2004)

[19] P. Kumar, "Cryptography with Java", http://www.informit.com/articles/article.asp?p=170967&seqNum=11, May 28 2004 (as of November 21st 2004)

[20] KeyTool - Key and Certificate Management Tool, http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html, (as of November 21st 2004)

[21] Ethereal help - 4.5 Filtering while capturing, http://ethereal.planetmirror.com/docs/user-guide/ChCapCaptureFilterSection.html, (as of November 21st 2004)

[22] Class System, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/System.html#currentTimeMillis(),(as of November 21st 2004)

[23] KeyStore Explorer, http://www.lazgosoftware.com/kse/, (as of November 21st 2004)

[24] Portecle, http://portecle.sourceforge.net/, (as of November 21st 2004)

[25] J. Ly, N. Williams, "Secure Public Instant Messaging (IM) at Work", July 2004

[26] CryptoHeaven, http://www.cryptoheaven.com, 2004 (as of November 21st 2004)

[27] Top Secret Manager (TSM), http://www.encrsoft.com/products/tsm.html, 2004 (as of November 21st 2004)

[28] V. Kumar, S. Doraiswamy, Z. Jainullabudeen, "Elliptic Curve Cryptography", March 31 2004

XIV. APPENDICES

A. *Cryptographic Primitives*

Cryptographic algorithms can be grouped into classes known as cryptographic primitives. There are three primitives: unkeyed, symmetric keyed and asymmetric keyed.

1. Unkeyed Primitives

Unkeyed algorithms do not use any keys in the generation of their output. An example of this type of algorithm is arbitrary length hash functions, the main subclass of unkeyed algorithms. Hash functions generate a hash value known as a "hash" or "message digest". The input of a hash function is just the message to be hashed. Hash functions are used in are one of the most versatile cryptographic algorithms and are used for many purposes such as being part of digital signature and key establishment schemes.

Other types of unkeyed algorithms include one-way permutations and random sequences. Unlike hash functions, these functions on their own do not provide any sort of cryptographic utility, however they can be implemented as part of the mathematical computations involved in other cryptographic algorithms.

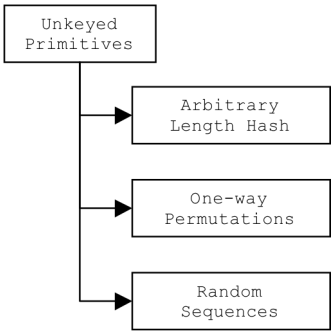

Figure 30 - Unkeyed Cryptographic Primitives

2. Symmetric Key Primitives

Symmetric key algorithms use a single key as one of its inputs to manipulate data. The keys used in these functions are shared only with authorised entities and kept secret from everyone else. Hence, such algorithms are also known as secret key algorithms.

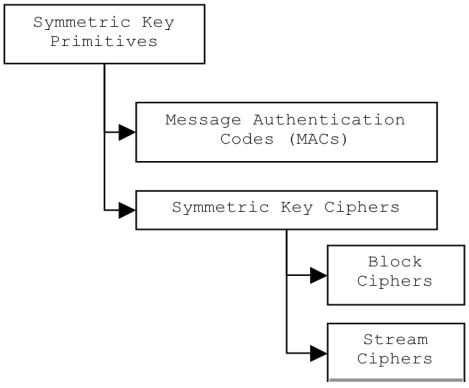

Figure 31 - Symmetric Key Cryptographic Primitives

Symmetric key algorithms have numerous applications such as providing confidentiality, data integrity and authentication as well as form part of a key establishment process.

There are two major types of symmetric key algorithms: Message Authentication Codes (MACs) and symmetric key ciphers. The primary purpose of MACs is to provide data origin authentication. MACs are also able to detect message tampering, thus providing data integrity checks. A MAC function takes a secret key and an arbitrary sized message and generates a fixed sized MAC value. This MAC value is sent with the encrypted message. The receiver generates a MAC value of the encrypted message and checks to see whether it matches the MAC value that was sent with the message. If there is a match, then the message has not been tampered with.

Symmetric key ciphers can be further broken down into to classes: stream ciphers and block ciphers. Block ciphers operate by breaking up an arbitrarily sized plaintext message into fixed sized blocks. These blocks are encrypted one at a time. There are three main classes of block ciphers: substitution, transposition and product ciphers. Substitution block ciphers substitute symbols or groups of symbols with other symbols or groups of symbols within a block. Transposition ciphers perform permutations on the symbols within a block. Alone, both substitution and transposition ciphers provide low levels of security. To overcome this, the operations of substitution and transposition ciphers are combined. This is what a product cipher is.

Stream ciphers can be thought of as a type of block cipher whereby the block size is one. They operate quicker than block ciphers and have the advantage of not propagating transmission errors. The main disadvantage of stream ciphers is that although stream cipher theory has been studied intensively, most stream ciphers implemented in current systems are proprietary and therefore secret. This leads to a lack of official documentation and standardised code libraries for stream ciphers.

3. Asymmetric Key Primitives

Asymmetric key algorithms use a set of two related keys known as a key pair. A key pair consists of a public key and a private key. This is why asymmetric key algorithms are also known as public key algorithms. A private key is only known to the entity that owns the key pair whereas a public key can be known to anyone, regardless of whether they are authorised by the owner entity. The important point is that the relationship between the public and private keys is such that the private key cannot be ascertained through knowing the public key.

An advantage of such algorithms is that the key exchange does not have to be done over a secure channel. However, it is necessary to authenticate the public keys

through data origin authentication to provide assurance that the owner of the public key is the intended communication partner. This is done through key establishment techniques, such as the one outlined in VI.E.

Asymmetric key algorithms are normally used in calculating digital signatures, establishing cryptographic material and for authentication schemes.

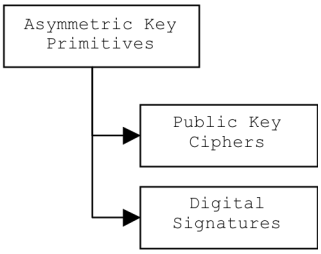

Figure 32 - Asymmetric Key Cryptographic Primitives

B. *Advanced Encryption Standard (AES)*

The Advanced Encryption Standard (AES) is a US government standardised block cipher. It was standardised through NIST, who had asked for submissions from the cryptographic community for a new block cipher algorithm. Out of the 15 submissions, the Rijndael algorithm was chosen to become AES. The Rijndael algorithm specifies a symmetric block cipher that processes block lengths of 128 bits using symmetric keys with sizes of 128, 192 or 256 bits. Rijndael was designed to handle other block sizes and key lengths, however these were not implemented as part of the AES standard.

One round of AES encryption is shown in Figure 33 - One Round of AES. The plaintext message is broken into blocks of 128 bits. Each block is encrypted separately. AES consists of 10-14 rounds, with each round using a round key derived from the symmetric key and one block of the plaintext message. The number of rounds depends on the size of the key. Each round can be thought of as a weak block cipher.

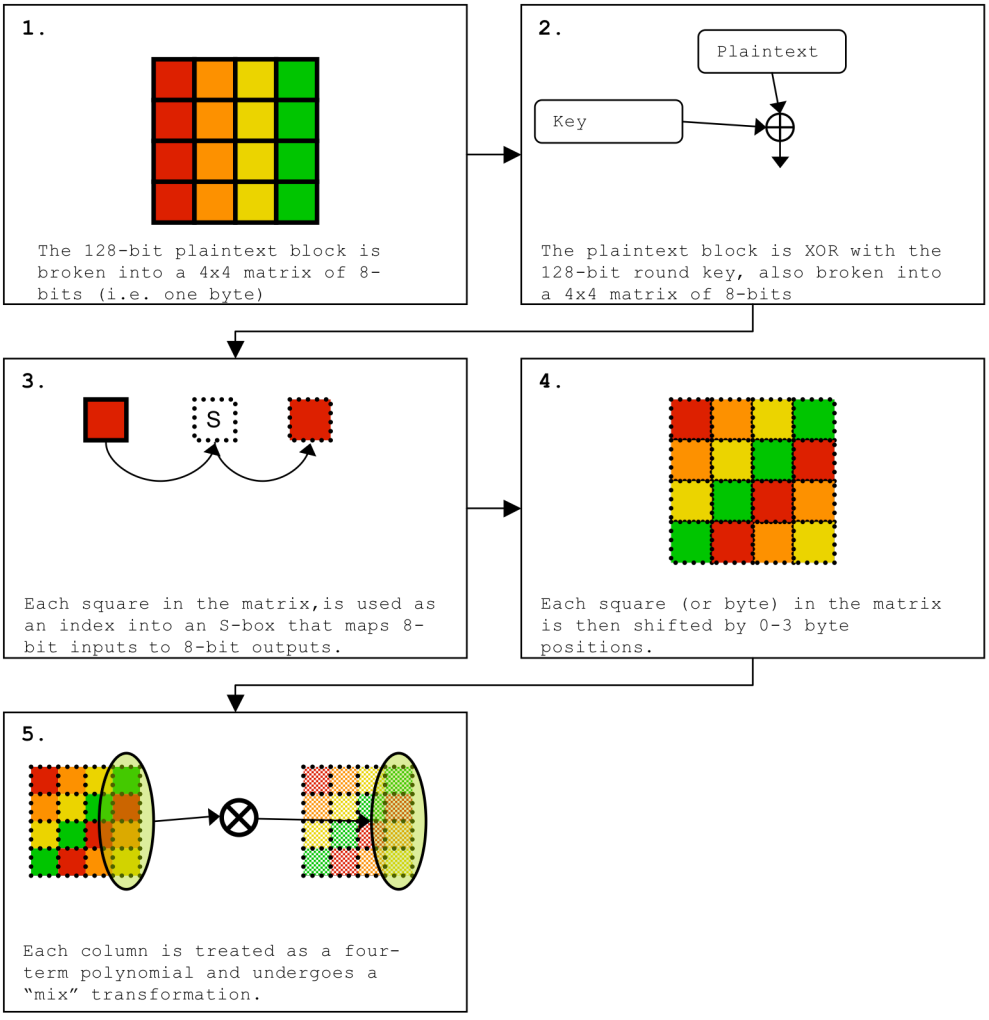

Figure 33 - One Round of AES

The first step is to break the 128-bit plaintext block into 16 bytes. The 16 bytes can be thought of as a 4x4 matrix of one byte (8 bits) called a State array. The round key is also broken up in the same manner and then XOR with the State array.

Each byte is used as an index into an S-box (substitution box), which maps the input byte to an output byte value. The transformation operation used in this step is a non-linear byte transformation that operates independently on each input byte. The S-boxes used are identical and its contents are publicly known (Figure 34 - S-box Used in AES Encryption).

The State array is then put through a row shifting transformation. Each row, other than the first row, is shifted cyclically over an offset. The offset depends on the row number. For example, the first row is row number 0, so it is not shifted. The second row is row number 1, so it is cyclically shifted by one position.

The final step is to perform a mixing transformation on the State array. Each column is treated as a four term polynomial and multiplied by a fixed polynomial.

This process is repeated for 10-14 rounds. In the final round, instead of the mixing transformation, the round key is XOR once again with the State array. This makes the algorithm reversible during decryption. The total number of rounds depends on the size of the key. Table 3 - Key-Block-Round Combinations lists the number of rounds in relation to the key size.

	Key Length (in words)	Block Size (in words)	Number of Rounds
128-bit key	4	4	10
192-bit key	6	4	12
256-bit key	8	4	14

Table 3 - Key-Block-Round Combinations

Each of the transformations are invertible, therefore similar steps can be used to perform the decryption, but in a reverse order. The decryption process consists of an initial round plus nine rounds of transformations, totalling in the ten rounds required for a 128-bit key. The initial decryption round consists of the following steps:

1. XOR round key
2. Inverse byte substitution transformation
3. Inverse row shifting transformation

The next nine rounds consists of the following steps:

1. XOR round key
2. Inverse mixing column transformation
3. Inverse byte substitution transformation
4. Inverse row shifting transformation

The inverse byte substitution involves using an S-box with the contents being the inverse of the contents of the S-box used in encryption (Figure 35 - Inverse S-box Used in AES Encryption). The inverse row shifting operation involves the same step as that in the encryption transformation, except that the rows are shifted to the left. The first row in the multiplication matrix is changed for the inverse mixing column transformation. Other than that, the mixing column transformation executes in the same manner as in the encryption process [10].

1. Key Expansion

To generate the round keys from the symmetric key, a method called the key expansion is used. The key expansion operation is performed before encryption begins. The symmetric key must be expanded to be long enough to provide enough keying material for the multiple round keys. Each round key is made up of a different section of the expanded symmetric key. The number of round keys need to be one more than the number of rounds as the round key is used twice in final round. Therefore, the total number of bytes required is:

```
16 x (number of rounds + 1)
```

When a 128-bit (16 byte) key is used, the expanded key must be 1408 bits (176 bytes) long. The symmetric key always makes up the first bytes in the expanded key. So for a 16 byte key, the first 16 bytes of the expanded key is the symmetric key.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 34 - S-box Used in AES Encryption

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 35 - Inverse S-box Used in AES Encryption

C. *Collision Attacks*

A collision occurs when a function provides the same output for two different input values. When a collision occurs, it can possibly leak information to an adversary such as what secret key is currently being used. Collision attacks are a type of cryptographic attacks that take advantage of the fact that there is a certain probability that collisions will occur in a cryptographic system after a certain period of time. There are two main types of collision attacks: birthday and meet in the middle.

1. Birthday Attacks

Birthday attacks make use of the fact that collisions occur in a shorter time period than would be expected. It is named after the birthday paradox, whereby the probability of two people having the same birthday out of a group of 23 people exceeds 50%. This is a high probability when the fact that there are 366 possible birthdays is taken into consideration.

An example of how this attack works is where a 64-bit key is used for a particular operation, it is expected that there are 2^{64} (18 billion) possible key values to select from. Therefore, it would seem that the attacker would have a difficult time trying to figure out the key being used. In actual fact, the attacker could expect to see the same key being used after approximately 2^{32} operations. If the attacker can ascertain that the same key is being used, then the system is susceptible to possible attacks, namely data insertion of old messages from the attacker.

In general, if there are n different possible values, then it is expected that the first collision will occur after approximately $\sqrt{n}$ random elements have been chosen. The birthday bound, which is related to this hypothesis, defines the fact that a collision is expected to occur after $2^{n/2}$ elements have occurred.

2. Meet in the Middle Attacks

A more common collision attack is the "Meet in the middle" attack. The way this type of attack works is that instead of waiting for a collision to occur, a set of keys can be randomly generated by the attacker and used as a reference when the attacker eavesdrops on a communication exchange. For example, in a situation where a MAC is used and the system uses the same first message to the user (such as a welcome message or a confirmation request), the attacker can randomly choose a set of keys and generate MAC values for the first message using these keys. The MAC values are stored in a table. In the event that the same MAC value is sent over the channel the attacker is listening in on, then it is highly probable that the key that is being used in the communication exchange is the same key that the attacker used to generate the MAC value.

This allows the attacker to insert any messages that the attacker chooses to generate into the communication exchange instead of just replaying old messages like with birthday attacks. This makes meet in the middle attacks more useful than birthday attacks. However, for both birthday and meet in the middle attacks, a collision can be expected to occur within the same number of elements.

D. Cipher Block Chaining (CBC)

The Cipher Block Chaining (CBC) mode is the most commonly used mode in current systems. It is a type of confidentiality mode where the encryption method involves "chaining" plaintext blocks to previous cipher text blocks.

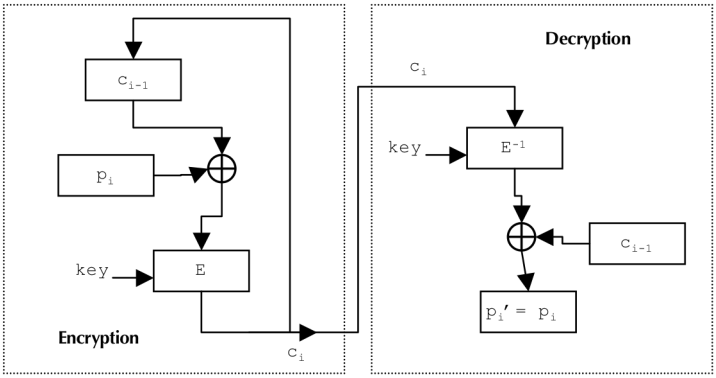

Figure 36 - CBC Encryption and Decryption

The CBC encryption function (Figure 36 - CBC Encryption and Decryption) uses an input of the previous ciphertext block c_{i-1}, where $i \geq 0$. This is then XOR with the plaintext block p_i. The result is inputted into the cipher function E (in the case of the project, E is AES) to produce the ciphertext c_i. As each CBC encryption cycle requires the previous ciphertext as an input, the encryption operation cannot be processed in parallel.

The CBC decryption function uses the ciphertext c_i as the input into the inverse cipher function E^{-1}. This is then XOR with the previous ciphertext block. The operation produces the output p_i' which should be equal to the plaintext message p_i. As the input of the decryption function (the ciphertext c_i) is immediately available, there can be multiple decryption operations processed in parallel.

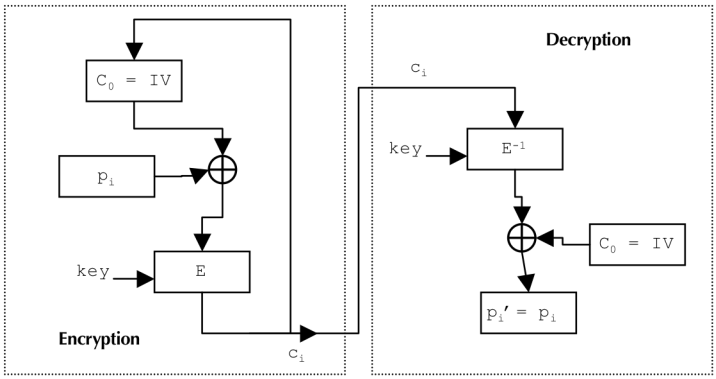

Figure 37 - CBC Encryption and Decryption for the First Cycle

In the case of the first block, as there is no previous ciphertext block, an IV is used instead. Therefore, c_0 is the IV.

The advantage of CBC is that it does not have a problem when encrypting two plaintext blocks that are the same. In ECB, if two plaintext blocks are the same, then the resultant ciphertext block will also be the same. CBC solves this problem by chaining the previous ciphertext block to the plaintext block, so that the plaintext blocks that are the same will be chained to different ciphertext blocks.

For the plaintext message to be recovered successfully, the sender and receiver need to have their IVs synchronised. The IVs used should not be the same as the problem with ECB is introduced for the first block of each message. As messages often start in a similar or identical manner, the message blocks would be the same (or similar). If the same IV is used then it is possible for the same ciphertext block to be generated. This provides information to the attacker. How IVs can be generated is discussed in Initialisation Vector (IV) Generation Methods.

E. Initialisation Vector (IV) Generation Methods

There are several ways to generate the IV value used in CBC. These include counter IV, random IV and nonce-generated IV.

1. Counter IV

A counter IV uses zero for the first message, one for the second message and so on. The main disadvantage of using this method is that numbers in sequence have a similar sequence when represented in binary format. As many messages start in a similar manner, the IVs could cancel out the minor differences between the starting messages, as the IVs are also similar. For example, if the IVs differ in exactly one bit and the start of two messages only differ in one bit then the ciphertext blocks may be identical due to the IVs cancelling out the differences. Even if the differences are not cancelled out, if the differences are only minor, an attacker can still use cryptanalysis to make presumptions about the messages being exchanged.

2. Random IV

Another solution to the IV problem is to use a random IV. As stated in the CBC section, the sender and receiver need to have their IVs synchronised to successfully recover the plaintext message. Consequently, the first ciphertext block (c_0) is the random IV value. Because the first block does not contain any plaintext, the resultant ciphertext block would be one block longer than the plaintext, with the extra block being the random IV block. This is a major disadvantage if the messages are short as it results in significant message expansion. Another disadvantage is that the encryption algorithm will need a source of randomness, which requires a lot of overhead if a good random generator is to be implemented.

3. Nonce-generated IV

The term nonce is derived from the phrase *n*umber used *once*. The nonce is a unique number that is not used twice with the same encryption key. The nonce is normally a message number. Since most communication systems require a message number to be implemented, using a nonce will not create any message overheads. The IV is then generated by encrypting the nonce with the block cipher implemented in the system.

The following steps describe how a nonce-generated IV is used in CBC:

1. A message is assigned a message number. Normally the message number is generated by a counter, starting at zero.
2. The message number is used as the unique nonce.
3. The IV is generated by encrypting the nonce with the block cipher (in the case of the project, the block cipher is AES).
4. CBC mode is then used to encrypt the message, using the nonce-generated IV.
5. Additional information is added to the ciphertext, such as attaching the message number to the ciphertext. This is to ensure that the receiver can generate the same nonce. The IV is not sent with the message.
6. Ensure that the receiver accepts one message at a time by rejecting any messages with a message number that is less than or equal to the message number of the last received message.

F. RSA

The RSA algorithm was developed by Ronald L. Rivest, Adi Shamir, and Leonard Adleman in 1977 and published in 1978. It is a commonly used cryptosystem that provides both encryption and digital signature functionalities.

1. Key Generation

The following steps outline the basic operations involved in generating an RSA key pair.

1. Find two large (say, between 1024 and 4096 bits) prime numbers, p and q. Use these numbers to compute $n = pq$.
2. Choose an exponent e where e is greater than one, an odd number and less than n. Also, e and $(p-1)(q-1)$ need to be such that they do not have any prime factors in common.
3. Calculate the multiplicative inverse of e, known as d. This can be done through finding an integer x that produces an integer result for the equation $d = (x(p-1)(q-1))$.

The ***public key*** is the pair (n, e). Hence, e is known as the *public exponent*. The public key can be known by anyone as there is currently no known way to derive d, p or q through knowing the public key (n, e). The ***private key*** is d. Consequently, d is known as the *private exponent*. It is imperative that d be kept secret from other entities.

2. Public Key Encryption

The encryption function for RSA is $c = m^e \bmod n$, where c is the ciphertext and m is the plaintext message. Both the ciphertext and the plaintext are positive integers. The value of m must be less than n. It can be seen that for RSA encryption, the public key is used.

For RSA decryption, the function is fundamentally the same, except that the private key is used instead of the public key. The decryption function is $p = c^d \bmod n$.

3. Digital Signatures

Generating digital signatures in RSA involves the same calculations used in encrypting and decrypting. To sign a plaintext message, the entity that owns the private key computes a signature $s = m^{1/e} \bmod n$. The signed message is therefore (m, s). This signature can be verified by any entity that knows the corresponding public key. This is done using the function $s^e = m \bmod n$.

G. *KeyTool*

1. About Key Tool and Important Terms

KeyTool is a Java command line utility that is used to create, manage and secure asymmetric keys and certificates in Java KeyStore files. Public keys can be exported from the KeyStore as one of a number of certificate types. Public keys of other persons can be imported into a KeyStore using the KeyTool utility.

There are a number of important terms in regards to items and processes when using the Java KeyTool. An *alias* refers to the name given to a particular item, such as a public key, within the KeyStore. A *store* is the name used to access the keystore, which is the name of the keystore file. A *provider* is the cryptographic suite that provides the encryption algorithms for the keystore.

Although the following sections provide some examples of using keytool via command line, a number of programs exist that provide a graphical interface for creating and editing keystore files. These programs are the KeyStore Explorer [23], which can be purchased from $US30 for a single user license, and Portecle [24] can be downloaded for free.

2. Generating RSA Key Pairs

The Bouncy Castle Cryptographic Provider was used to generate the RSA keys required for encryption. This provider was used as the Sun Provider as in JDK1.4.2 did not contain support for the RSA algorithm.

To generate an asymmetric key-pair the following command was entered from the console:

```
keytool -genkey -alias <alias> -keyalg RSA -
keysize 1024 (or 2048) -sigalg MD5withRSA -
keystore <storename>.ks -storepass <password> -
storetype jks
```

Keytool then provides prompts for additional information, such as the name of the key-pair owner and the expiry date of the key. This information can also be entered in the initial command if desired.

3. Importing X.509/CER Files

Public keys are exchanged as specially encoded certificate files. These files can be exchanged via disk, email or via the 'export public key' function in the add-on. If using email or the export function, it is important to verify the fingerprint of the certificate. See KeyTool for an example of how this is done.

To import a key file, for example "alicecert.cer" into Bob's keystore, the following command would be used:

```
Keytool –import –alias alice –file alicecert.cer –
keystore bobstore –storepass password
```

A prompt then asks whether this certificate is to be trusted, to which 'yes' is entered. The public key is now stored within Bob's keystore with the alias "alice".

4. Verifying Received Certificate

It is recommended that public key certificates be exchanged in person using a diskette rather than through the export function of the add-on. If this is not possible, however, it is necessary to manually check the MD5 fingerprint of the received key to ensure it has not been subject to any man-in-the-middle attack.

After importing the certificate, the fingerprint can be obtained by typing the command:

```
Keytool –list –keystore <storename> -storepass
<password>
```

This lists the certificates contained within the keystore and their MD5 fingerprint. Imported certificates will appear as '`trustedCertEnrty`', while private keys will appear as '`keyEntry`'. The example below shows the keyEntry fingerprint for a certificate.

```
nigel, 3/09/2004, keyEntry,
Certificate fingerprint (MD5):
41:3A:0E:27:E1:A5:E2:F0:07:2B:EC:D2:88:03:48:2E
```

The received public key certificate contains the same fingerprint as the keyEntry certificate. Therefore, the fingerprint should be confirmed (over the phone, for instance) by comparing the fingerprint of the received certificate against the certificate contained in the original keyEntry.

H. System Clock Granularity

For example, a Windows XP system may have a clock granularity of 15-16ms and would report the following times if it were polled on a 10ms basis.

System Time	Poll Interval
06:05:14.000	0ms
06:05:14.000	10ms
06:05:14.015	20ms
06:05:14.030	30ms
06:05:14.030	40ms
06:05:14.046	50ms
06:05:14.060	60ms

Table 4 - System Clock Granularity

In the above example, the clock granularity is larger than the polling interval. Thus, when the first 10ms interval has elapsed, the clock is still reporting the time `06:05:14.000` – it has yet to be updated. After 20ms have elapsed, the clock has been updated to reflect a time of `06:05:14.015`. After 30ms the polling interval falls in line with the clock update, and the time is reported accurately. At 40ms, the clock has yet to refresh, thus the time is still reported as `06:05:14.030`. 50ms shows that the interval was 16ms, as the reported time is `06:05:14.046`. The final poll at 60ms also coincides with the clock update, giving a system time of `06:05:14.060`.

As can be seen, the accuracy of `currentTimeMillis()` can vary significantly over short timed intervals. It could be said that the margin of error for the system above would be ± 16ms. For short period of time, such as 30ms, ± 16ms would represent a margin of error of over 50%. Intervals less than 15ms would simply appear as 0ms.

In contrast, ± 16ms of a 300ms operation roughly represents a 5% margin of error, which is much more acceptable than 50%. Therefore, where possible, all timed functions were placed into a loop to decrease the margin of error and hence increase the accuracy of the result.

I. Benchmark Tables of Results

The following tables contain the results of each trial conducted in the benchmarks. Adjusted figures indicate that the total time has been divided by the number of loops, giving the average time for an individual operation.

1. AES Benchmark Results

AES Key generation	
System	Time(ms)
A	110
	100
	110
C	30
	31
	30
D	661
	110
	110
E	80
	30
	20

Table 5 - AES Key Generation

AES 58-byte String, 50,000 loops		
System	Encrypt(ms)	Decrypt(ms)
A	200	161
	200	171
	200	160
C	250	211
	240	210
	241	210
D	961	811
	962	821
	962	821
E	160	140
	161	130
	160	140

Table 6 - AES Encryption and Decryption of 58-byte String

AES 108-byte String, 50,000 loops		
System	Encrypt(ms)	Decrypt(ms)
A	161	160
	171	160
	160	160
C	200	210
	210	201
	210	210
D	761	771
	751	781
	761	781
E	130	130
	130	130
	130	130

Table 7 - AES Encryption and Decryption of 108-byte String

AES 58-byte String Adjusted figures		
System	Encrypt(ms)	Decrypt(ms)
A	0.004	0.00322
	0.004	0.00342
	0.004	0.0032
C	0.005	0.00422
	0.0048	0.0042
	0.00482	0.0042
D	0.01922	0.01622
	0.01924	0.01642
	0.01924	0.01642
E	0.0032	0.0028
	0.00322	0.0026
	0.0032	0.0028

Table 8 - AES Encryption and Decryption Adjusted Figures (58-byte)

AES 108-byte String Adjusted figures		
System	Encrypt(ms)	Decrypt(ms)
A	0.00322	0.0032
	0.00342	0.0032
	0.0032	0.0032
C	0.004	0.0042
	0.0042	0.00402
	0.0042	0.0042
D	0.01522	0.01542
	0.01502	0.01562
	0.01522	0.01562
E	0.0026	0.0026
	0.0026	0.0026
	0.0026	0.0026

Table 9 - AES Encryption and Decryption Adjusted Figures (108-byte)

2. RSA Benchmark Results

RSA Benchmark 500 Loops time in milliseconds					
System	Sign	encode public	decode private	verify signiture	decode public
A	11566	491	6640	7310	400
	11496	491	6620	7260	390
	11797	531	6670	7300	400
B	29952	1443	14381	15983	951
	29688	1444	14382	15985	950
	29664	1456	14362	15993	945
C	16053	751	10586	10895	641
	15713	681	10044	10656	601
	15472	681	9894	10666	591
D	56602	2844	32857	36102	2123
	56632	2613	33098	37834	2233
	59095	2955	34289	37624	2214
E	10946	531	7210	7831	430
	10976	500	7231	7811	441
	10913	521	7200	7882	441

Table 10 - RSA Benchmark Results (500 Loops)

RSA Benchmark adjusted time in milliseconds					
System	Sign	encode public	decode private	verify signiture	decode public
A	23.132	0.982	13.28	14.62	0.8
	22.992	0.982	13.24	14.52	0.78
	23.594	1.062	13.34	14.6	0.8
B	59.904	2.886	28.762	31.966	1.902
	59.376	2.888	28.764	31.97	1.9
	59.328	2.912	28.724	31.986	1.89
C	32.106	1.502	21.172	21.79	1.282
	31.426	1.362	20.088	21.312	1.202
	30.944	1.362	19.788	21.332	1.182
D	113.204	5.688	65.714	72.204	4.246
	113.264	5.226	66.196	75.668	4.466
	118.19	5.91	68.578	75.248	4.428
E	21.892	1.062	14.42	15.662	0.86
	21.952	1	14.462	15.622	0.882
	21.826	1.042	14.4	15.764	0.882

Table 11 - RSA Benchmark Results (Adjusted Figures)

J. *Selected Prototype Source Code*

The source code in the following sections is from the main prototype java classes.

1. SortIM

```
/* This will take an Instant Message which has been given an escape sequence
   and identify the command code given to allow processing as appropriate. */

/*
 * sortIM.java
 *
 * Created on 25 April 2004
 */

package au.edu.swin.jn;

import com.tomjudge.TjMSN.*;
import com.tomjudge.TjMSNLib.*;

import java.io.*;
import java.util.*;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.spec.*;
import javax.crypto.*;
import javax.swing.*;
import java.security.Key;
import java.security.cert.*;

public class sortIM
{
    private String trimMessage;
    private String cmdCode;
    private String codedMsg;
    private String toEncode;
    private String toDecode;
        private String decodedMessage;

        private String storeName;
        private String storepass;
        private String aliasLocal;
        private String aliasRemote;

        private boolean sendReply;
        private boolean encryptionStatus;
        private boolean endEncryptSent = false;

        String AESkey = null;
        String CBCkey = null;
        String decodedPrivate;
        int messageCounter = 0;
        newAlias getAliasDialog;

//    byte[] theKey = genKey();
    AEScrypt encrypter;
    AEScrypt decrypter;
    AESGen keyGen;
    RSAmanager test;


/* Command Code summery
*
*   Code            Function
*   11             Request encrypted session
*   12             Receive request for encrypted session
*   13             Acceptance of request
*       14              Recieve Acceptance
*       15              Send signature
*       16              Recieve signature
*   21             Message to be encrypted and transmitted
*   22             Message to be decrypted
*   23             Part 1 of split message for re-assembly and decryption
*   24             Part n of split message
*   25             Final part of split message
```

```
    *   30              End Session command
    *       31              Recieve End Session command
    *       40              Message part 1 of cert
    *       41              recieve part one
    *       42              cert message part 2
    *       43              recieve part two
    */

       public sortIM(boolean status)
       {
              encryptionStatus = status;
           }

           public String sort(String aMessage) { //make a public variable for aMessage and use that for
decrypt?
               if (encryptionStatus)
               {
                   if (aMessage.startsWith("###@ "))
                   {
                       cmdCode = aMessage.substring(5,7);

                       if (cmdCode.matches("21"))
                       {
                           activeEncryption();
                           toEncode = aMessage.substring(8);
                           encrypter = new AEScrypt(AESkey, messageCounter); // also want to pass
messagecount to create the IV
                           trimMessage = toEncode; //for history pane
                           String ciphertext = encrypter.encrypt("###@ 22 " +toEncode); // Encrypt the
message for transmission

                           codedMsg = encrypter.getMac(CBCkey, ciphertext)+ciphertext;
                           messageCounter++; //increment the counter
                       }

                       else if (cmdCode.matches("30")) //sending
                       {
                           activeEncryption();
                           trimMessage = "notice to de-activate encryption";
                           encrypter = new AEScrypt(AESkey, messageCounter);
                           String ciphertext = encrypter.encrypt("###@ 31 ");
                           codedMsg = encrypter.getMac(CBCkey, ciphertext)+ciphertext;
                           endEncryptSent = true;
                        }
                       else if (cmdCode.matches("33"))
                       {
                           if (endEncryptSent) {
                           noEncryption();
                           trimMessage = "encryption has been de-activated";
                           sendReply = false;
                           messageCounter = 0; }
                       }

                   }
                   else
                   {
                       decrypter = new AEScrypt(AESkey, messageCounter); //(theKey, IV)
                       String recievedMac = aMessage.substring(0,8);
                       boolean result = decrypter.verifyMac(CBCkey, aMessage.substring(8), recievedMac);
                       if (result) {
                       decodedMessage = decrypter.decrypt(aMessage.substring(8));
                       cmdCode = decodedMessage.substring(5,7); }
                       else
                           {
                       decodedMessage = "###@ 22 Bad MAC message discarded";
                       cmdCode = decodedMessage.substring(5,7);
                       }
                       messageCounter++;

                       if (decodedMessage.startsWith("###@ "))
                       {
                           if (cmdCode.matches("22"))
                           {
                               activeEncryption();
                               trimMessage = decodedMessage.substring(8); //decoded msg for display
```

```
                            sendReply = false;
                        }
                        else if (cmdCode.matches("31")) //recieving
                        {
                            noEncryption();
                            trimMessage = null;
                            codedMsg =  "###@ 32 ";
                            sendReply = true;
                            messageCounter = 0;
                        }
                    }
                }
             }
             else
             {
                 cmdCode = aMessage.substring(5,7);
                 if (cmdCode.matches("11")) //request for encryption
            {
                    noEncryption();
                    trimMessage = "you have requested an encrypted session";
                    keyGen = new AESGen();
                    AESkey = keyGen.genKey();
                    CBCkey = keyGen.getCBCKey();
                    test = new RSAmanager(storeName, storepass, aliasLocal, aliasRemote); //to replace
with variables
                    String encmsg = test.encodeMessagePublic(CBCkey+AESkey);
                    codedMsg = "###@ 12 " +encmsg; //AESkey variable replaced by the result of public
key encryption

            }
            else if (cmdCode.matches("12")) //recieve request for enc
            {
                    noEncryption();
                    trimMessage = "wishes to start an encrypted session";

                    test = new RSAmanager(storeName, storepass, aliasLocal, aliasRemote); //to replace
with variables
                    decodedPrivate   =   test.decodeMessagePrivate(aMessage.substring(8));   //take   off
cmdCode, decode and save AESkey
                    AESkey = decodedPrivate.substring(24);
                    CBCkey = decodedPrivate.substring(0,24);
                    //no need for anything else, the ###@ 16 will directly follow
                    sendReply = false;
            }
                 else if (cmdCode.matches("13")) // sending a 13
                 {
                     //set the variable to say that encryption is enabled
                     noEncryption();
                     test = new RSAmanager(storeName, storepass, aliasLocal, aliasRemote);
                     String signed = test.signMessage(CBCkey+AESkey);
                     codedMsg = "###@ 14 "+signed;
                     trimMessage = "awaiting confirmation";
                 }
            else if (cmdCode.matches("14")) //recieving a 14
                 {
                     //set the variable to say that encryption is enabled
                     noEncryption();
                     test = new RSAmanager(storeName, storepass, aliasLocal, aliasRemote);
                     test.verifyMessage(decodedPrivate, aMessage.substring(8));
                     boolean theResult = test.verResult();

                     if (theResult)
                     {
                         int r = JOptionPane.showConfirmDialog(null, "Enable encrypted session?");
                         if (r == JOptionPane.YES_OPTION)
                         {
                             codedMsg = "###@ 15 ";
                             trimMessage = null;
                             sendReply = true;
                         }
                         else
                         {
                             codedMsg = "the request has been denied";
                             trimMessage = "you denied the request";
                             sendReply = true;
```

```
                        }
                    }
                    else
                    {
                        trimMessage = "signature did not match";
                        codedMsg = "The signature did not match, request cancelled";
                        sendReply = true;
                    }

                }
                else if (cmdCode.matches("15")) //sending a 15
                {
                    activeEncryption();
                    codedMsg = "###@ 16 ";
                    trimMessage = "Encryption activated";
                }
                else if (cmdCode.matches("16")) //receiving
                {
                    activeEncryption();
                    trimMessage = "request accepted. session encrypted";
                    sendReply = false;
                }
                else if (cmdCode.matches("41")) //sending
                {
                    RSAmanager rsaMan = new RSAmanager(storeName, storepass, aliasLocal, aliasRemote);
                    codedMsg = "###@ 42 " +rsaMan.exportLocalCert();
                    trimMessage = "you have exported your public key";
                }
                else if (cmdCode.matches("42")) //recieving
                {  /* MUST CHECK TO ENSURE THAT KEYSTORE HAS BEEN CONFIGURED */

                    try {
                    byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(aMessage.substring(8));
                    getAliasDialog = new newAlias(null, true);
                    getAliasDialog.show();
                    String userAlias = getAliasDialog.getNewAlias();
                    if (userAlias != null) {
                        /* READING IN AND CREATING THE CERT */
                        ByteArrayInputStream bais = new ByteArrayInputStream(dec);
                        BufferedInputStream bis = new BufferedInputStream(bais);
                        CertificateFactory cf = CertificateFactory.getInstance("X.509");
                        X509Certificate cert = null;
                        while (bis.available() > 0) {
                              cert = (X509Certificate)cf.generateCertificate(bis);
                        }
                              byte[] buf = cert.getEncoded();
                              FileOutputStream os = new FileOutputStream(userAlias+".cer");
                              os.write(buf);
                              os.close();
                              trimMessage = "the certificate has been saved as: " +userAlias +".cer. Use
keytool -import -alias <alias> -file <file.cer> -keystore <keystore> -keypass <keypass> to import the
file.";
                    }
                    else {
                    trimMessage = "not imported";
                    }

                }
                 catch (Exception e) {
                    e.printStackTrace();
                    }

                }
                else if (cmdCode.matches("32"))
                    {
                        codedMsg = "###@ 33 ";
                        trimMessage = "Encryption has been de-activated";
                    }
            }
            return null;
        }

        public String getTrim()
        {
                return trimMessage;
```

```
    }

    public String getCoded()
    {
            return codedMsg;
    }

        public void activeEncryption()
        {
            encryptionStatus = true;
        }

        public void noEncryption()
        {
            encryptionStatus = false;
        }

        public boolean getEncryptionStatus()
    {
            return encryptionStatus;
    }

        public boolean sendReplyMsg()
        {
            return sendReply;
        }

        public void setKeystore(String store_name, char[] store_pass)
        {
            storeName = store_name;
            storepass = new String(store_pass);
        }

        public void setAlias(String local_alias, String remote_alias)
        {
            aliasLocal = local_alias;
            if (remote_alias != null) {
                aliasRemote = remote_alias;
            }

        }
}
```

2. AESCrypt

```
/*
 * AEScrypt.java
 * This class looks after the creation of AES key, and also performs the
 * encryption and decryption of message.  TODO: The IV should be obtained using
 * the supplied message number. (unless random IV is used)
 * Created on 9 September 2004, 13:21
 */
package au.edu.swin.jn;

import java.lang.*;
import javax.crypto.KeyGenerator;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.Key;
import java.security.NoSuchAlgorithmException;
import java.security.Security;
import java.security.*;
import org.bouncycastle.crypto.macs.*;
import org.bouncycastle.crypto.engines.*;
import org.bouncycastle.crypto.params.*;

/**
 *
 * @author  nigel
 */
public class AEScrypt {
    Cipher ecipher;
    Cipher dcipher;
    Cipher IVcipher;
```

```
        SecretKeySpec skeySpec = null;
        SecretKeySpec CBCSpec = null;
        IvParameterSpec ips;
        CBCBlockCipherMac macHash;
        boolean macResult;

        /** Creates a new instance of AEScrypt */
        public AEScrypt(String keyString, int messageCounter) {
            try {


                byte[] keyAgain = new sun.misc.BASE64Decoder().decodeBuffer(keyString);  //base64 key string
to byte
                skeySpec = new SecretKeySpec(keyAgain, "AES");                           //make a usable key
object from this

                ecipher = Cipher.getInstance("AES/CBC/PKCS7Padding","BC");
                dcipher = Cipher.getInstance("AES/CBC/PKCS7Padding","BC");
                IVcipher = Cipher.getInstance("AES","BC");

                String ivString = Integer.toBinaryString(messageCounter);
                byte[] iv = ivString.getBytes();
                IVcipher.init(Cipher.ENCRYPT_MODE, skeySpec);
                byte[] encIV = IVcipher.doFinal(iv);
                ips = new IvParameterSpec(encIV);

                AESEngine a = new AESEngine();
                macHash = new CBCBlockCipherMac(a,32);

            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }

        /* TODO: Perhaps use counters so that the IV need not be sent. */

        public String encrypt(String str) {
        try {
     //     ecipher.init(Cipher.ENCRYPT_MODE, skeySpec,secRandom);
            ecipher.init(Cipher.ENCRYPT_MODE, skeySpec, ips);
            byte[] utf8 = str.getBytes("UTF8");                 //get UTF8 bytes from String
            byte[] enc = ecipher.doFinal(utf8);                 //do the encryption
            return new sun.misc.BASE64Encoder().encode(enc);    //return base64 encrypted string
        }
        catch (Exception e) {
                e.printStackTrace();
        }
        return null;
        }

        public String decrypt(String str) {
        try {
            String encString = str;
    //      dcipher.init(Cipher.DECRYPT_MODE, skeySpec, new IvParameterSpec(aes_iv));
            dcipher.init(Cipher.DECRYPT_MODE, skeySpec, ips);
            byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(encString);      //Base64 String to byte
            byte[] decoded = dcipher.doFinal(dec);                                  //decrypt the byte
            return new String(decoded, "UTF8");                                       //Convert byte to UTF8
string
        }
        catch (Exception e) {
                e.printStackTrace();
            }
        return null;
        }
        public String getKey()
        {
            String keyString = new sun.misc.BASE64Encoder().encode(skeySpec.getEncoded());
            return keyString;
        }
        public String getMac(String CBCkey, String message)
        {
            try {
            byte[] CBCAgain = new sun.misc.BASE64Decoder().decodeBuffer(CBCkey);
```

```
        CBCSpec = new SecretKeySpec(CBCAgain, "AES");

        byte[] stringBytes = message.getBytes();
        byte[] hash = new byte[4];
        macHash.init(new KeyParameter(CBCSpec.getEncoded()));
        macHash.update(stringBytes,0,stringBytes.length);
        int blah = macHash.doFinal(hash,0);
        macHash.reset();
        return new sun.misc.BASE64Encoder().encode(hash);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
    public boolean verifyMac(String CBCkey, String message, String theMac)
    {

        try {
            byte[] CBCAgain = new sun.misc.BASE64Decoder().decodeBuffer(CBCkey);
            CBCSpec = new SecretKeySpec(CBCAgain, "AES");

            byte[] stringBytes = message.getBytes();
            byte[] hash = new byte[4];
            macHash.init(new KeyParameter(CBCSpec.getEncoded()));
            macHash.update(stringBytes,0,stringBytes.length);
            int blah = macHash.doFinal(hash,0);
            macHash.reset();
            String newMac = new sun.misc.BASE64Encoder().encode(hash);
            if (theMac.equals(newMac)) {
               macResult = true; }
            else {
                macResult = false; }
            }
        catch (Exception e) {
            e.printStackTrace();
        }
        return macResult;
    }
}
```

3. RSAManager

```
/*
 * RSAmanager.java
 *
 * Created on 2 September 2004, 13:32
 */

package au.edu.swin.jn;

import java.lang.*;
import java.io.*;
import javax.crypto.*;
import java.security.Provider;
import java.security.cert.*;
import java.security.*;
/**
 *
 * @author  nigel
 */
public class RSAmanager {

    char[] ksPass = null;
    KeyStore ks = null;
    PrivateKey privKey = null;
    X509Certificate remote_cert = null;
    X509Certificate local_cert = null;
    String local_alias = null;
    String remote_alias = null;
    String storepass;
    byte[] theSig = null;
    boolean testResult = false;

    KeyPair pair;
```

```
    /** Creates a new instance of RSAmanager */
    public RSAmanager(String keystore_file, String store_pass, String aliasLoc, String aliasRem) {
        try {
        local_alias = aliasLoc;
        remote_alias = aliasRem;
        storepass = store_pass;

        Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
        ks = KeyStore.getInstance("JKS","SUN");
        ks.load(new FileInputStream(keystore_file), storepass.toCharArray());
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
    /* take an MD5 of message and ecnrypts with a private key */
    public String signMessage(String theMessage) {
        try {
            Signature signature = Signature.getInstance("MD5WithRSA");
            signature.initSign((PrivateKey) ks.getKey(local_alias, storepass.toCharArray()));

            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            DataOutputStream dos = new DataOutputStream(baos);

            String sr = theMessage;
            byte[] utf8 = sr.getBytes("UTF8");
            dos.write(utf8);
            //dos.writeUTF("a string of sorts");
            byte[] arrayBytes = baos.toByteArray();

            ByteArrayInputStream bais = new ByteArrayInputStream(arrayBytes);
            DataInputStream dis = new DataInputStream(bais);

            byte[] buffer = new byte[arrayBytes.length];
            int length;
            while ((length = dis.read(buffer)) != -1)
            signature.update(buffer, 0, length);
            dis.close();

    //      dis.read(buffer, 0, buffer.length);
    //      System.out.println(dis.readUTF());
    //      System.out.println(new String(buffer, "UTF8"));

            byte[] raw = signature.sign();
            theSig = raw;
            return new sun.misc.BASE64Encoder().encode(raw);

        }
        catch (Exception e)
            {
                e.printStackTrace();
            }
        return null;
    }
    /* takes the signature string and tries to verify it */
    public void verifyMessage(String theMessage, String theSignature) {
        try {

            Signature signature = Signature.getInstance("MD5WithRSA");
            signature.initVerify(ks.getCertificate("Nigel").getPublicKey());

            /* read in the message file */
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            DataOutputStream dos = new DataOutputStream(baos);

            String sr = theMessage;
            byte[] utf8 = sr.getBytes("UTF8");
            dos.write(utf8);
            //dos.writeUTF("a string of sorts");
            byte[] arrayBytes = baos.toByteArray();

            ByteArrayInputStream bais = new ByteArrayInputStream(arrayBytes);
            DataInputStream dis = new DataInputStream(bais);
```

```
            byte[] buffer = new byte[arrayBytes.length];
            int length;
            while ((length = dis.read(buffer)) != -1)
            signature.update(buffer, 0, length);
            dis.close();

            /* Convert Signature String to Bytes and Test */
            byte[] signedBytes = new sun.misc.BASE64Decoder().decodeBuffer(theSignature);

            if (signature.verify(signedBytes)) {
                testResult = true;
            }
            else{
                testResult = false;
                }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public boolean verResult() {
        return testResult;
    }

    public String encodeMessagePublic(String theMessage)  {
        try {
        remote_cert = (X509Certificate) ks.getCertificate(remote_alias);
        PublicKey pKey = remote_cert.getPublicKey();
   //   CertificateFactory cf = CertificateFactory.getInstance("X.509");
        Cipher RSAcrypt = Cipher.getInstance("RSA", "BC");
        RSAcrypt.init(Cipher.ENCRYPT_MODE,pKey);

        byte[] utf8 = theMessage.getBytes("UTF8");
        byte[] enc = RSAcrypt.doFinal(utf8);
        return new sun.misc.BASE64Encoder().encode(enc);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public String decodeMessagePrivate(String theMessage) {
        try {
        Cipher RSAdecrypt = Cipher.getInstance("RSA","BC");
        RSAdecrypt.init(Cipher.DECRYPT_MODE,(PrivateKey)                              ks.getKey(local_alias,
storepass.toCharArray()));
        byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(theMessage);
        byte[] decoded = RSAdecrypt.doFinal(dec);
        return new String(decoded, "UTF8");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public String encodeMessagePrivate(String theMessage) {
        try {
        Cipher RSAencrypt = Cipher.getInstance("RSA","BC");
        RSAencrypt.init(Cipher.ENCRYPT_MODE,(PrivateKey)                              ks.getKey(local_alias,
storepass.toCharArray()));
        byte[] utf8 = theMessage.getBytes("UTF8");
        byte[] enc = RSAencrypt.doFinal(utf8);
        return new sun.misc.BASE64Encoder().encode(enc);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public String decodeMessagePublic(String theMessage)  {
```

```
        try {
        remote_cert = (X509Certificate) ks.getCertificate(remote_alias);
        PublicKey pKey = remote_cert.getPublicKey();
        Cipher RSAdecrypt = Cipher.getInstance("RSA", "BC");
        RSAdecrypt.init(Cipher.DECRYPT_MODE, pKey);
        byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(theMessage);
        byte[] decoded = RSAdecrypt.doFinal(dec);
        return new String(decoded, "UTF8");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public String exportLocalCert() {
        try {
        local_cert = (X509Certificate) ks.getCertificate(local_alias);

        byte[] certBytes = local_cert.getEncoded();
        return new sun.misc.BASE64Encoder().encode(certBytes);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
   }
    public void importCert(X509Certificate theCert, String theAlias) {
        try {

            byte[] buf = theCert.getEncoded();
            FileOutputStream os = new FileOutputStream(theAlias+".cer");
            os.write(buf);
            os.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```