ROYAL HOLLOWAY, UNIVERSITY OF LONDON

Enigma: Prime Numbers and Cryptosystems

by

Adam Mulligan

A final year **full-unit** project submitted in partial fulfillment for the degree of Bachelor of Science, Computer Science

in the
Department of Computer Science
Royal Holloway, University of London, TW20 0EX

March 2012

“Anyone can design a security system that he cannot break. So when someone announces, “Heres my security system, and I can‘t break it, your first reaction should be, Who are you? If he‘s someone who has broken dozens of similar systems, his system is worth looking at. If he‘s never broken anything, the chance is zero that it will be any good.”

Bruce Schneier, *The Ethics of Vulnerability Research*

Abstract

A vital part of modern-day communication is the ability to converse with confidence that you are not being eavesdropped on – a confidence that is ensured through the use of digital cryptography. Contemporary cryptography provides end-to-end solutions for this, including key-agreement, encryption, and certification, most of which are based on fundamental number theory problems.

This report discusses the process of understanding, creating and using these solutions in a fully-fledged application and any implications and issues that arise.

Following this guide will provide with all the information necessary to understand and create a secure instant messaging system with the option of implementing multiple cryptographic algorithms for all information security requirements.

Acknowledgements

With thanks to my project advisor Yuri Kalnishkan, and the RHUL Department of Computer Science for three years worth of knowledge and experience.

Contents

Chapter 1

Introduction

1.1 What it is about

Secrecy has always been of great importance, not just in modern society, but throughout history. Until very recently, Cryptography was consigned to the sending and receiving of messages, generally using pen and paper. However, increasingly cryptography – the study and implementation of techniques for secure communication – is becoming more vital to the smooth running of even basic systems, whether it is the cliché example of military secrets or simply a micro-payment for an online service. Initially, the usage of provably secure and efficient cryptographic algorithms was limited to governments and related contractors, however the advent of cryptographic standards, and more relevantly the creation of public-key cryptography mechanisms, has pushed it in to a wider field of use as researchers gained a better understanding of the area.

1.2 Goals and Intentions

The primary goal of this project can be found in the title and abstract: to research, discuss and create algorithms within or related to the field of number theory. To complete this task I will be developing my thoughts and discoveries regarding number theory as this report progresses, as well as producing a number of deliverables in the form of software applications, test results and statistics.The topic of number theory is in itself fascinating, and extraordinarily vast, however I will only be scratching the surface. Nonetheless, I hope to explain throughout this project how number theory has become such an important part of modern cryptography, and what it can be used for.

1.3 Project Summary

I will be splitting the project in to three main parts, excluding this report:

1. **Prime numbers in software**
   A discussion of how prime numbers, and an application of number theory, can be efficiently produced programmatically, and the software to prove it.

2. **Algorithms**
   The development of algorithms using prime numbers, such as RSA, and complementary cryptographic algorithms such as AES. Alongside this, the development of non-major algorithms in the field of prime number based cryptography that still present academically interesting concepts.
3. **Final Application**
   The production of a software program that utilises the implemented cryptographic algorithms in §2 to display their efficacy in a real world application.

The primary algorithms to develop are:

- Asymmetric
  - RSA
  - Diffie-Hellman
- Symmetric
  - AES
- Identification and Authentication
  - RSA Signing
  - Digital Certificates

The reasoning behind this selection is thus: popularity. The best algorithms are those that are mature and have been scrutinised thoroughly by an open source community. However, we will further discuss this choice as the report progresses. Other algorithms will be discussed and produced alongside these, however they will not be used in the final application testbed and are purely for research interest. These can be found listed in the contents in the relevant sections.

1.4 Other Information

This document was written and composed with LaTeX using *Taco Software's* Latexian[1]. The LaTeX source code for this report should have come bundled with the project documentation and files, however if you are unable to retrieve it please see §*1.4.2*.

Eclipse Indigo[2] was used for Java development, the primary language used in this project, along with *MacVim*[3] for general source and text editing.

Git with *GitHub*[4] was used for source control, with *GitHub Issues* used for bug tracking. *Trello*[5] was used for idea and to-do management. *OmniGraffle*[6] was used to produce graphics, such as sequence diagrams and system overviews.

[1] http://tacosw.com/latexian/
[2] http://eclipse.org
[3] http://code.google.com/p/macvim/
[4] http://github.com
[5] http://trello.com
[6] http://www.omnigroup.com/products/omnigraffle/

1.4.1 A note on openness

As with any field of study, the quality of research and development is dependent on the open distribution and sharing of ideas. This is *particularly* important with regards to cryptography. As said, cryptography was once reserved to government and research was conducted in secrecy. The open sharing of relevant information in this field is not just for the furthering of knowledge, but also to allow others to inspect and examine algorithms, a process that drastically improves the security of a system. As such, the entirety of this project is licensed under the **GNU Lesser General Public License version 3 or greater** and is available for access publicly online.

1.4.2 Project Repository

If any part of this report is missing, you believe that you do not have a full access to the source code discussed in this document, or if any files have been lost, it is available for full download at `http://cyanoryx.com/files/project.tgz`. It is also available at `http://github.com/adammulligan/Enigma`.

Throughout the report we reference files by their package locations. For example, *com.cyanoryx.uni.crypto.rsa.RSA_OAEP* signifies that the file is located at *src/com/cyanoryx/uni/crypto/RSA_OAEP.java.*

Chapter 2

Cryptographic Primitives

2.1 Basics of Information Security

Despite how it is portrayed or colloquially used, Information Security is an entirely different concept and area of study compared to Cryptography. It might seem simple enough to implement a basic cryptographic protocol involving encryption and decryption, however to introduce this into a system and expect the information to be secure, is foolhardy. Cryptography is a *means* to providing information security when following certain rules and guidelines and is not the be all, end all solution. An understanding of information security, and the related issues, is necessary.

This can be proven using historical evidence: throughout history many complex systems of mechanisms, rules, and protocols have been developed to introduce information security to a system. As with modern day security, this cannot be achieved entirely through mathematical and cryptographic means – it is more than just computational intractability.

As such, stringent criteria for developing secure systems and protocol have been introduced. While institutes such as *The British Computing Society* and *Association for Computing Machinery* ensure their members follow a professional code of ethics, just as a doctor might, these information security criteria are of separate and equal importance. Indeed, there are now several international organisations that exist solely for overseeing cryptographic research and development (See: *International Association for Cryptologic Research*).

Often, as we will see, cryptographic systems are simplified for the purposes of presentation particularly for textbooks. This is problematic and will be discussed further later, with regards to the differences and difficulties involved in developing systems that do not just follow a mathematical "recipe," but also include information security values and other subtleties.

The overall method of dealing with, and ensuring, information security is known as risk management. This encapsulates a large number of countermeasures (including cryptography) that reduce the risk of vulnerabilities in, and threats to, systems. We will only be encountering and discussing the technological areas of mitigation, however some of the solutions include[1]: access control, security policy, physical security, and asset management.

2.2 Objectives

As said, secure systems should follow a guideline, or set of criteria, that ensure the security and integrity of data stored and input. A clear and concise specification should be developed, that will aid the designer in selecting the correct cryptographic primitives, but also help the engineer implement the protocol correctly. There are many of these criteria, however each is derived from four primary objectives:

1. **Confidentiality** is the ability to ensure data is only accessed by those who are allowed to. Maintaining confidentiality of data is an obligation to protect someone else's secret information if you have been entrusted with it.
2. **Authentication** involves identifying both entities and data. Two or more entities wishing to communicate or transmit information to one another must identify each participant to ensure they are who they claim to be - this is known as entity authentication. Data received must be authenticated to ensure the validity of the origin, date sent, contents, etc - this is known as data origin authentication.
3. **Non-repudiation** prevents an entity from denying previous actions they have committed.
4. **Data Integrity** is how faithfully data compares to its true state, i.e. proving that a data object has not been altered.

2.3 Primitives

As we discussed in §2.2, there are certain criteria that must be met for an application to be considered as secure under Information Security guidelines. Excluding physical and psychological measures, there are a number of methods to be implemented cryptographically to guarantee security.

2.3.1 Encryption

Being what is seen as the very 'core' of cryptography, we have defined encryption many times already and what the term means in terms of a process should be apparent. However encryption takes many forms, with each having an appropriate situation for it to be used.

[1] For an excellent resource regarding information security, both technical and non-technical, see [1]

2.3.1.1 Symmetric Key Encryption

Primarily, we will use symmetric key encryption algorithms to encipher data that is to be transmitted between entities.

Mathematically we can formally define symmetric encryption as:

For a message M, algorithm A and key K,

$M' = A(K, M)$

and thus:

$M = A'(K', A(K, M))$

where $A' = A$, $K' = K$ in a symmetric algorithm.

2.3.2 Key Agreement

Key agreement, or key exchange, primarily ensures data integrity and confidentiality. By preventing an attacker from discovering encryption keys used on transmitted data, the attacker should be unable to feasibly read confidential information or modify it (the latter is not entirely true, it may be possible in some cases to modify encrypted data, however we will discuss that later in §4).

While it might be easier to use asymmetric techniques to encrypt data for transmission, thus allowing us to distribute keys as cleartext, it is slow and inefficient for large quantities of data, such as in an instant messaging application. Because of this, it is prudent to implement an efficient symmetric key encryption algorithm, and share the key (known as a session key) with other entities. However, it would be trivial for an attacker to launch a man-in-the-middle attack and gain access to the encryption keys during the initiation of the conversation, allowing the easy and undetectable decryption of all transmitted messages. As such, session keys will need to be distributed using a key exchange protocol.

2.3.2.1 Key Distribution Centre

The simplest solution is known as a Key Distribution Centre (KDC), which involves the use of a trusted third party (TTP). It is easiest to explain using an example. Alice and Bob are users of a system, attempting to securely communicate. Each share a key securely with third-party Trent (somewhat amusingly, this algorithm does not include how these keys should be shared. We can assume that it was perhaps conducted through an in-person meeting of entities, or other means), who stores each key.

1. Alice initiates a conversation with Bob.
2. Alice requests a session key from Trent, who makes two copies of an identical key and encrypts one with Alice's stored key, and another with Bob's.
3. Alice receives both encrypted keys, and sends the appropriate one to Bob.

4. Alice and Bob decrypt their session keys, leaving both with a shared key.
5. Alice and Bob can now encrypt and decrypt data using the same key.

2.3.2.2 Asymmetric Cryptography

As can be obviously seen, using a KDC is dependent entirely on the ability of two entities to have previously, and securely, shared an encryption key with a TTP that is known to both entities. A solution to this is to eliminate the third party, and use an asymmetric algorithm to share keys directly. Asymmetric cryptography allows Alice to share a public-key, with which any entity can encrypt data that can only be decrypted using Alice's private key.

1. Alice and Bob share their public keys using a readily available database.
2. Alice downloads Bob's key, and vice versa.
3. Alice generates a session key, encrypts it using Bob's public key and sends it to Bob.
4. Bob can now decrypt the session key using his private key, resulting in both parties being in possession of a secure session key.

There is a security risk: how can you verify that the entity that sent the encrypted key is indeed the one with which you are trying to communicate? It would be easy for an attacker to encrypt their own session key with Alice's public key, and claim that the key is from Bob. The attacker would then be able to decrypt any messages intended for Bob. The solution to this issue is the use of digital certificates, and signatures, which will be discussed in §5.

We will discuss specific algorithms further on, however it is worth pointing out that the current most-used public-key algorithm used for these purposes is *RSA* (Rivest, Shamir, Adleman) as it is the cipher used in the handshaking process for Transport Layer Security in networking. There exist a number of interesting algorithms that implement the public key architecture which will also be considered later.

There are other considerations in key management to ensure confidentiality and integrity. Some provisos exist such as changing the key for each session to ensure perfect forward secrecy, however these are trivial to implement and can be considered as part of the overall application security development.

2.3.3 Authentication

There are two types of authentication that are required in a secure application

1. Message authentication – this allows an entity to send a message using a method that allows the receiver to detect any changes to the message while in transit.
2. Entity authentication – this allows an entity to verify that the person sending or receiving messages to or from them are indeed who they claim to be.

both of which require different protocols and algorithms.

The methods of entity authentication are an interesting topic in themselves, with many, many protocols having been researched and created as the community tries to find a method that is both secure and easy for an entity to use. Some examples are: passwords, two-factor authentication, PINs, smart cards, biometrics, and so on. These are outside of the scope of this project – we will be implementing two broad, yet specific methods of authentication.

2.3.4 Digital Signatures

Digital signatures, as we will see, encompass three of the information security criteria: authentication, data integrity and non-repudiation (a sender cannot claim they did not send the message). A digital signature is a string that connects a message with its originating entity.

When transmitting a message, the sender signs the message with their private key which can then be verified with their public key, which should be available to the receiver.

1. Alice signs her message A with her private key.
2. Alice sends message A with signature S to Bob $\{A, S\}$
3. Bob retrieves Alice's public key from an available database, and verifies signature S

The first and most common implementation of digital signatures is RSA.

2.3.5 Public-key Certificates

Digital signatures and a public-key infrastructure, however, are not enough by themselves. A very simple attack can be orchestrated similar to that during key agreement: Mallory, the attacker, could sign a modified message A with her own private-key and then distribute her public-key, claiming it to be Alice's. To counter this, we can introduce a trusted third-party, known as a Certificate Authority (CA), who signs Alice's public-key with their own private-key. Most trusted CA's public keys come bundled with software such as browsers and operating systems.

1. The Certificate Authority signs Alice's public key with their private key.
2. The CA distributes its public key with major software, such as browsers and operating systems.
3. Bob receives Alice's message and signature, as well as her public key and signature.
4. Bob verifies Alice's public key with the CA's public key, and then verifies the message.

2.3.6 Hashing

While not of direct relevance in information security, hashing plays a significant part in cryptographic systems and thus is included as a concept to be implemented. Formally, we can define a hash function as mapping a large domain to a smaller range - in the case of data, mapping a set of bytes to a unique identifier with a set length. A hash function, at the very basics, takes as input a message and produces a fingerprint, or digest, of the input. Within the field of cryptography, they are used for message authentication and data integrity.

This is to say, given a domain D and range R for $f : D \rightarrow R$, then $|D| > |R|$. This is a many-to-one relationship, the downside of which means that collisions can occur – two input strings resulting with the same output string – however, this varies between algorithms, the more modern of which are less likely to result in collisions.

A hash function can be classed into two categories: keyed and unkeyed, taking both a message and secret key and taking just a message, respectively. Two conditions are necessary for a hash function to be effective:

1. compression – the function f maps input a of arbitrary length to an output of fixed length, n.
2. complexity – it must be easy to compute $f(a)$

Most commonly used are unkeyed, one-way hash functions. Some examples of which are: SHA-1, and MD5. In cryptography, hashes are commonly used for data integrity in combination with digital signatures. A message is hashed, and the fingerprint produced is signed by the entity. There are some algorithms designed specifically for this purpose, known generally as Message Authentication Codes (MACs) and Manipulation Detection Codes (MDCs).

A sample of the first three primitives will be implemented further in the report.

2.3.7 Complexity

Computational complexity is a vast subject, mostly out of the scope of this report, however we will use the notation occasionally to classify algorithms. Big-oh notation, as it is known, will be used to represent the worst-case running time of an algorithm based on a standard input size. For example:

Definition 2.3.1. A polynomial-time algorithm is an algorithm where the worst-case running time can be represented as $O(n^c)$, where n is the input size, and c is a constant.

An algorithm with a running time that cannot be bounded as such is known as an exponential-time algorithm. It is generally considered that polynomially-time algorithms are efficient, wheres exponential-time algorithms are inefficient.

Computational complexity will not be used extensively this report, except in algorithm comparison, and as such only a basic knowledge of its notation is expected.

2.4 Moving On

This is just a basic overview of the cryptographic primitives used in Information Security. As the report progresses, each concept will be explained in finer detail alongside their implementations. While it seems hard to apply these abstract definitions, the purpose of each within our algorithms will become quickly clear.

Chapter 3

Number Theory and Public-key Cryptography

3.1 Overview

Number theory, the unit of mathematics that studies integers and their properties, was once a predominantly useless area. However, as cryptographic algorithms became more prevalent, particularly schemes that require the use of large prime numbers, so too did number theory. We will cover the notation used in the majority of number theoretic algorithms, and will build upon this and discuss basic concepts and applications: modular arithmetic, prime number generation, and factorisation.

In this section we will make the assumption that any attackers who may try to break our algorithms or eavesdrop on our messages are extremely powerful and capable, and so we will discuss algorithms with regards to their tractability – in this case, whether or not a significant percentage of all instances of a problem can be solved in polynomial time.

Throughout, where appropriate, programming examples are provided in Java.

3.2 Mathematics

This section will cover some of the basic mathematical concepts that will be used throughout the report, and form a foundation of understanding for the more complex abstract methods that will be used.

3.2.1 Notation

1. $\mathbb{Z}$ is the set of all integers.
2. $\mathbb{R}$ is the set of all real numbers.
3. $[a, b]$ is the set of integers n such that $a \leq n \leq b$.

4. $|A|$ is the cardinality of a finite set, i.e. the number of elements.
5. $n \in A$ denotes that an element n exists in set A.
6. $A \subseteq B$ denotes that set A is a subset of set B.
7. $A \subset B$ denotes that A is a subset of B, but $A \neq B$.
8. $\lceil a \rceil$ is the smallest integer greater than or equal to a.
9. $\lfloor a \rfloor$ is the largest integer less than or equal to a.
10. A function (mapping) denoted as $f : A \rightarrow B$ signifies that every element a in A is mapped to exactly one element b in B. This can also be written as $f(a) = b$.

3.2.2 Number Theory

We will begin by defining some fundamental rules of number theory.

Definition 3.2.1. Let a, b be integers. a divides b if an integer c exists such that $b = ac$. We define this as a divides b, or $a|b$.

Definition 3.2.2. An integer c is a common divisor if $c|a$ and $c|b$.

Definition 3.2.3. An integer c is the greatest common divisor of a and b, if:

1. c is a common divisor of a, b.
2. when $d|a$ and $d|b$, $d|c$.
3. $c \geq 0$

This is denoted as $c = \gcd(a, b)$.

Definition 3.2.4. An integer a is prime if:

1. $a \geq 2$
2. the only positive divisors are 1 and a.

Otherwise, the number is referred to as composite.

Fact 3.2.1. There are an infinite number of prime numbers.

Definition 3.2.5. If a is prime and $a|bc$, then $a|b$ and $a|c$.

Definition 3.2.6. Integers a and b are relatively prime (coprime) to one another, if $\gcd(a, b) = 1$.

Definition 3.2.7. The function ϕ is known as the Euler Phi function. Where $n \geq 1$, $\phi(n)$ is the number of integers in the interval $[1, n]$ that are relatively prime to n.

1. If n is prime, then $\phi(n) = n - 1$.
2. $\phi(n)$ is multiplicative: if $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m) \cdot \phi(n)$.

3.2.2.1 Modulo Arithmetic

Definition 3.2.8. Where a and b are integers, a is congruent to b modulo n (denoted as $a \equiv b \mod n$. n is known as the modulus. Congruence is reflexive, transitive and and symmetric. That is to say, for every $a, b, c \in \mathbb{Z}$:

1. $a \equiv a \mod n$
2. if $a \equiv b \mod n$, then $b \equiv a \mod n$
3. if $a \equiv b \mod n$ and $b \equiv c \mod n$, then $a \equiv c \mod n$

This is known as an equivalence relation.

Definition 3.2.9. The set of all integers modulo n is denoted as $\mathbb{Z}_n$.

3.3 Modular Arithmetic and Congruence

As we know, the core concept of modular arithmetic is to return the remainder of the division of two integers, represented as $a(mod\ n)$. It is said that b is congruent to $a(mod\ n)$ if $b \equiv a(mod\ n)$. Congruences play a great part in cryptography – one of the very first uses was the Caesar Cipher. Messages were made secret – encrypted – by shifting letters forward by an integer n, where $0 < n \leq 25$.

Given a sequence of characters $\{A, B, C, ..., Z\}$ where each letter is represented by a number $\{0, 1, 2, ..., 25\}$. Defined by a function $f(n)$, where n is the integer representation of a character:

$$f(n) = (n + x)\ mod\ 26$$

where x is the number of characters to shift by.

This is a very simple example of how congruences can be used in cryptography. As we will see in the public-key section, congruences can be applied in a similarly simple way as a vital component part of a far more complex algorithm.

Indeed, although the technicalities of this are not relevant, congruences can even be used to generate pseudo-random numbers.

3.4 Prime Numbers

Primes have already been covered, however they are a deeply fascinating subject with many current open problems and conjectures. For example, Goldbach's Conjecture states that every odd integer n, where $n > 2$, is the sum of two primes. This is intriguing because so far this has been *verified* for integers up to 2×10^{17}, but mathematicians have not yet found a formal proof for this, and despite this they continue to believe it to be true [2].

We, however, will be using them for a far more interesting[1] purpose: cryptography based around the intractability of factoring very large composite numbers into their smaller non-trivial prime divisors.

3.4.1 Generation

First and foremost, we must devise an efficient method for generating the prime numbers. Although generating the keys for use in prime number based algorithms is generally not done on a regular basis, it is still a complex task to complete in a reasonable amount of time. As it stands, a number of algorithms already exist (and have done for many hundreds of years) for this purpose, though in a different sense: it is relatively easy to generate small sequences of small primes, however to obtain large, individual primes the fastest method is to generate probable primes using primality tests.

To help understand these algorithms, we must first state a number of theorems that make clearer how these algorithms work and how they will fit in to the public-key implementations later on [3]:

Definition 3.4.1. We define $\pi(x)$ as the number of primes found in the interval $[2, x]$, where $\pi(x) \sim \frac{x}{\ln x}$.

Fact 3.4.1. If $gcd(a, n) = 1$, there are infinitely many primes that are congruent to a mod n.

Fact 3.4.2. We can say that all prime numbers are uniformly distributed across $\phi(n)$ because:

$$\pi(x, n, a) \sim \frac{x}{\phi(n) \ln x}$$

Where $\pi(x, n, a)$ is the number of primes in the interval $[2, x]$ congruent to $a \mod n$.

Fact 3.4.3. We can approximately determine the nth prime number as $\rho_n \sim n \ln n$, where $n \geq 6$.

3.4.1.1 Probable, Provable and Pseudo-primes

There is a subtle yet significant difference between the types of primes that can be generated.

Probable Primes A probable prime is a prime that meets certain conditions that are satisfied by all prime numbers. Tests for probable primes are more appropriately known as *compositeness tests* rather than probable primality tests.

The algorithms we will define as probabilistic primality tests will take arbitrary integers and test them to provide information about their primality.

[1] http://xkcd.com/247/

Pseudo-primes Pseudo-primes are a subset of probable primes. They are composite numbers that pass tests that most composite numbers fail, for example an integer is pseudo-prime if it satisfies Fermat's Little Theorem.

Provable Primes A provable prime is a prime that can be formally proven to be prime using an algorithm, such as the AKS primality test. Generally these are not used in cryptography due to the previous inefficiency of calculating provably-prime numbers, and also as they work most effectively when the input has been passed through a probabilistic primality test first.

3.4.1.2 Sieves

A prime sieve is a fast algorithm to find prime numbers, the most commonly used being the *sieve of Erastothenes* and *sieve of Atkin*. Prime sieves operate by generating a list of integers up to a defined limit n: $\{2, 3, 4, ..., n\}$ and progressively removing the composite integers in line with particular rules. However, this is very slow for the generation of individual large primes, and so is immediately discounted from possible use in a cryptographic algorithm.

3.4.1.3 Fermat's Little Theorem

Note: Fermat's theorem is no longer considered to be a true probabilistic primality test as it cannot determine the difference between probable primes and composite integers known as Carmichael numbers[2]. The theorem can still be used to prove the compositeness of a number.

Fermat's theorem states:

$$\text{If } n \text{ is prime and } a \text{ is an integer, where } 1 \leq a \leq n-1, \text{ then } a^{n-1} \equiv 1 \mod n$$

Using this, we can take an integer n and to test for primality find an integer in this interval where the equivalence is not valid, thus proving that n is composite, and is not likely to be prime.

The implementation of this in Java is as so:

```java
import java.math.BigInteger;
import java.util.Random;

public class FermatTesting {
  public static boolean checkPrime(BigInteger n, int iterations) {
    Random rng = new Random();

    if (n.equals(BigInteger.ONE)) return false;

    for (int i=0; i<iterations; i++) {
```

[2] Carmichael numbers are similar to Fermat primes as they satisfy the congruence $b^{n-1} \equiv 1 \mod n$. The conditions for a number to be in the Carmichael set are more complex than this, however the details are out of the scope of this section.

```
      // Create an integer within the interval [1,n-1]
      BigInteger a = new BigInteger(n.bitLength(),rng);
      while (BigInteger.ONE.compareTo(a) > 0 || a.compareTo(n) >= 0) {
        a = new BigInteger(n.bitLength(),rng);
      }

      // a^(n-1)
      // Repeated until a!=1, thus proving it cannot be prime
      a = a.modPow(n.subtract(BigInteger.ONE),n);

      if (!a.equals(BigInteger.ONE)) return false;
    }

    return true;
  }
}
```

This file can be found at latex_src/Code/primes/FermatTesting.java

This is a very simple algorithm to implement. It selects a random integer in the interval $[1, n-1]$ and computes a^{n-1}. If $a \neq 1$, then a must be composite.

Note: We are using the class `BigInteger` to represent integers, as it allows far larger bit representations of integers than Java's standard 64-bit primitive `integer` type. `BigInteger` offers all the same operators that can be used with `integer` primitives, alongside useful operations such as modular arithmetic, GCD and even built in primality testing. We will use `BigInteger` for all Java-based programming that involves large integers.

3.4.1.4 Miller–Rabin Primality Test

In practice, the Miller–Rabin test is the most used primality test, that also is based on a set of equivalences that are true for primes, and thus if a number to be checked does not pass these equivalences it is not prime.

Definition 3.4.2. Let n be an odd prime: $n-1 = 2^d r$, where d and r are positive integers, with r being odd. For $a \in (\mathbb{Z}/n\mathbb{Z})$:

$a^r \equiv 1 \mod n$ **or**
$a^{2^d r} \equiv -1 \mod n$ assuming $0 \leq d \leq s-1$.

Using this, we can define the Miller–Rabin requirements as:

1. Assume n is an odd composite integer, and $n-1 = 2^s r$, where r is odd.

2. Where a is an integer in the interval $[1, n-1]$, if a does not match the conditions in definition 3.3.2 ($a^r \not\equiv 1 \mod n$, $a^{2^s r} \not\equiv -1 \mod n$), then a is known as a witness for n.

3. Where a is the same as a in point 2, if a matches *either* of the conditions in definition 3.3.2, it is known as a strong liar for n.

When n is determined to be a composite, it is known as a *witness*, meaning it is a definite composite number. When we refer to probable primes, they are known as *strong liars* as they are a likely probably prime to the base integer – it is known as a “strong” liar due to the fact that n could still be composite while the prime equivalences still hold.

The implementation of this in Java is as so:

```java
import java.math.BigInteger;
import java.util.Random;

public class MillerRabinTest implements PrimeTest {
 public boolean checkPrime(BigInteger n, int iterations) {
  if (n.equals(BigInteger.valueOf(2L))) return true; // 2 is prime

  if (n.equals(BigInteger.ZERO) || // n==0
    n.equals(BigInteger.ONE)  || // n==1
    // n is even
    n.mod(BigInteger.valueOf(2L)).equals(BigInteger.ZERO)) {
   return false;
  }

  // 2 ^ s * r
  int s=0;
  BigInteger r,n_one;
  r = n_one = n.subtract(BigInteger.ONE);

  // Halve r until r is odd
  while (r.mod(BigInteger.valueOf(2L)).equals(BigInteger.ZERO)) {
   r = r.divide(BigInteger.valueOf(2L));
   s++;
  }

  for (int i=0;i<iterations;i++) {
   // Create a random number between 1 and n-1
   BigInteger a;
   do {
    int min = BigInteger.ONE.bitLength();
    int max = n_one.bitLength();
    a = new BigInteger(new Random().nextInt(max - min + 1)
             +min,new Random());
   } while (BigInteger.ONE.compareTo(a) > 0 || a.compareTo(n) >= 0);

   // y = a^r mod n
   BigInteger y = a.modPow(r,n);

   // While y != 1 and y != n-1
   if (!y.equals(BigInteger.ONE) && !y.equals(n_one)) {
    for (int j=0;j<s;j++) {
     // y = y^2 mod n
     y = y.modPow(BigInteger.valueOf(2L),n);

     // if y == 1, n is prime
     if (y.equals(BigInteger.ONE)) return false;

     // if y == n-1, n is composite
     if (y.equals(n_one)) return true;
    }
   }
  }

  return false;
 }
}
```

This file can be found at latex_src/Code/primes/MillerRabinTesting.java

In some cases the number of iterations is known as the *security parameter*.

3.4.1.5 Solovay–Strassen Primality Test

The Solovay–Strassen test was one of the first primality tests to be made popular by the increasing use of public-key cryptography. It is another probabilistic primality test. It is, in essence, quite a simple algorithm in that the process flow is linearly straightforward.

Definition 3.4.3. Let n be an odd integer, then $a^{\frac{(n-1)}{2}} \equiv \left(\frac{a}{n}\right) \mod n$ where $\gcd(a, n) = 1$.

We can use this to define the Solovay–Strassen algorithm requirements:

1. If $\gcd(a, n) > 1$ or $a^{\frac{(n-1)}{2}} \not\equiv \left(\frac{a}{n}\right) \mod n$, then a is an Euler witness for n (composite).
2. If $\gcd(a, n) = 1$ and $a^{\frac{(n-1)}{2}} \equiv \left(\frac{a}{n}\right) \mod n$ then n is an Euler pseudoprime (prime).

This is relatively simple to implement:

```
import java.math.BigInteger;
import java.util.Random;

public class SolovayStrassenTest implements PrimeTest {
  public boolean checkPrime(BigInteger n, int iterations) {
    BigInteger n_one = n.subtract(BigInteger.ONE);

    for (int i=0;i<iterations;i++) {
      BigInteger a = new BigInteger(n.subtract(BigInteger.valueOf(2L))
                                     .bitLength(),
                             new Random()).add(BigInteger.valueOf(2L));
      int x=jacobiSymbol(a,n);

      // r = a^(n-1)/2 mod n
      BigInteger r = a.modPow(n_one.divide(BigInteger.valueOf(2L)),n);
      // if (a|n)=0 or r!=x and r!=(n-1)
      if (x==0
          || (r.compareTo(BigInteger.valueOf(x))!=0)
          && (r.compareTo(n_one)!=0)) {
        return false;
      }
    }

    return true;
  }

  // Based on Algorith 2.149 in Alfred-Menezes:1996kx
  public static int jacobiSymbol(BigInteger a, BigInteger n){
      int j = 1;

      BigInteger ZERO  = BigInteger.ZERO;
      BigInteger ONE   = BigInteger.ONE;
      BigInteger TWO   = BigInteger.valueOf(2L);
      BigInteger THREE = BigInteger.valueOf(3L);
      BigInteger FOUR  = BigInteger.valueOf(4L);
      BigInteger FIVE  = BigInteger.valueOf(5L);
      BigInteger EIGHT = BigInteger.valueOf(8L);

      BigInteger res;

      while (a.compareTo(ZERO) != 0){
        // While a % 2 == 0
        while (a.mod(TWO).compareTo(ZERO) == 0){
          a = a.divide(TWO); // a / 2
          res = n.mod(EIGHT); // n % 8

          if (res.compareTo(THREE) == 0
              || res.compareTo(FIVE) == 0) j = -1*j;
        }

        BigInteger temp = a;
        a = n;
        n = temp;

        if (a.mod(FOUR).compareTo(THREE) == 0
            && n.mod(FOUR).compareTo(THREE) == 0) j = -1 * j;
        a = a.mod(n);
      }

      if (n.compareTo(ONE) != 0) j = 0;

      return j;
    }
}
```

This file can be found at latex_src/Code/primes/SolovayStrassenTest.java

Definition 3.4.4. $(\frac{a}{n})$ is known as the Jacobi symbol, and plays an important part in the Solovay–Strassen test. Let m, n be odd integers where $m \geq 3$ and $n \geq 3$, and $a, b \in \mathbb{Z}$, then we have the following properties[3]:

1. $(\frac{a}{n}) = 0$ or 1 or -1.
2. $(\frac{a}{n}) = 0$ if $\gcd(a, n) \neq 1$.
3. $(\frac{ab}{n}) = (\frac{a}{n})(\frac{b}{n})$
4. $(\frac{a}{nm}) = (\frac{a}{m})(\frac{a}{n})$
5. If $a \equiv b \mod n$ then $(\frac{a}{n}) = (\frac{b}{n})$. This is one of the most useful properties.
6. $(\frac{1}{n}) = 1$
7. $(\frac{-1}{n}) = (-1)^{\frac{n-1}{2}}$, and so $(\frac{-1}{n}) = 1$ if $n \equiv 1 \mod 4$, or $(\frac{-1}{n}) = -1$ if $n \equiv 3 \mod 4$.
8. $(\frac{2}{n}) = (-1)^{\frac{n^2-1}{8}}$, and so $(\frac{2}{n}) = 1$ if $n \equiv 1$ or $7 \mod 8$, or $(\frac{2}{n}) = -1$ if $n \equiv 3$ or 5 mod 8
9. $(\frac{m}{n}) = (\frac{n}{m})$ unless $m, n \cong 3 \mod 4$ which makes $(\frac{m}{n}) = -(\frac{n}{m})$.

The Java implementation of this is given above.

3.4.1.6 Lehmann Primality Test

A far lesser known primality test is Lehmann's Primality Test. Commonly (at least, within the minority that implement it) referred to as a balance between the simplicity of Fermat's Theorem and the complexity of implementation of Miller–Rabin, it consists of three steps:

1. Pick a random integer a, such that $1 \leq a < n$.
2. Let $x = a^{\frac{(n-1)}{2}} \mod n$.
3. If x is 1 or $-1 \mod n$ then is is a possible prime, otherwise it is composite.

The Java implementation of this is as so:

```java
import java.util.Random;
import java.math.BigInteger;

public class LehmannTest implements PrimeTest {
  public boolean checkPrime(BigInteger n, int iterations) {
    if (n.equals(BigInteger.ONE)) return false;

    for (int i=0;i<iterations;i++) {
      BigInteger a;
      do {
        // Pick a random integer a
        a = new BigInteger(n.bitLength(),new Random());
      // 1 <= a < n
      } while (BigInteger.ONE.compareTo(a) >= 0
               || a.compareTo(n) < 0);

      a = a.modPow(n.subtract(BigInteger.ONE)
                    .divide(BigInteger.valueOf(2L)),
                   n);

      // If x=1 or x=-1, then n is probably prime
      if (a.equals(BigInteger.ONE)
          || a.equals(BigInteger.valueOf(-1L).mod(n)))
        return false;
    }

    return true;
  }
}
```

This file can be found at latex_src/Code/primes/LehmannTest.java

We include this here as an example that subtle improvements to these algorithms can result in considerable improvements to running time[4] without adding to the complexity, though we will not consider it as a final algorithm.

However, simplicity does not come without a cost: as with Fermat's Theorem, it is susceptible to considering Carmichael numbers as possibly prime[3].

[3] http://news.ycombinator.com/item?id=3373284

3.4.1.7 AKS Primality Test

The only true-prime test of the four listed so far, AKS was presented as recently as 2002 by three mathematicians: Manindra **A**grawal, Neeraj **K**ayal and Nitin **S**axena (hence AKS) [5]. It is a deterministic primality-proving algorithm, with a complexity relatively similar to that of the faster probabilistic primality tests.

The AKS test is based around Fermat's Little Theorem, however it excels where Fermat's theorem was unable to: it is able to distinguish between pseudoprimes and Carmichael numbers. Recall that Fermat's Little Theorem states that:

Fact 3.4.4. Where integers $a \in \mathbb{Z}$, $n \geq 2$ and $\gcd(a, n) = 1$, n is prime if the following congruence is satisfied:
$(x + a)^n \equiv x^n + a \mod n$

This fails for both pseudoprimes and Carmichael numbers. Instead, the following congruence is used as the base of the AKS algorithm:

Definition 3.4.5. Where r is the smallest possible value that satisfies $O_r(n) > 4log^2n$,
$(x + a)^n \equiv x^n + a \mod x^r - 1, n$

The algorithm for AKS is fairly simple to follow, however the Java program is somewhat verbose, and thus it is helpful if we define it in pseudocode (uses the Lenstra and Pomerance Improvements [6]):

```
1. If $n = a^b$ where $a \in \mathbb{N}$ and $b > 1$ then print ”composite” and stop.
2. Find the smallest value of $r$ so that $O_r(n) > 4log^2n$.
3. If $\gcd(a, n) \neq 1$ for all integers $a$ where $a \leq r$ then print ”composite” and stop.
4. For $(a = 1)$ to $\lfloor \sqrt{r} \log n \rfloor$:
5. If $((x + a)^n) \equiv x^n + a \mod (x^r - 1, n)$ then print ”prime” and stop.
```

To calculate r, we use the following method:

```
1. Pick a number $q$ where $q > \lfloor log^2n \rfloor$.
2. For $j = 1$ to $[(log^2n)]$ do
3. Calculate $n^j \mod q$
4. If the residue $= 1 \mod q$, then $q++$.
5. Else $r = q$.
```

3.4.1.8 Maurer's Algorithm

Maurer's Algorithm is the second of two provable prime tests. It is an efficient recursive algorithm, that is only marginally less efficient than the generation of *probable primes* using Miller–Rabin.

First, let's define a number of facts necessary for this algorithm[7].

Definition 3.4.6. Let $n = 2RF + 1$ where the prime factorisation of $F = q_1^{b_1}...q_r^{b_r}$. Where a is an integer such that:

$$a^{n-1} \equiv 1 \mod n \text{ and } \gcd(a^{\frac{(n-1)}{q_r}} - 1, n) = 1$$

Then, a prime factor p of n is $p = mF + 1$ for an integer $m \geq 1$.
If $F > \sqrt{n}$, or F is odd and $F > R$, then n is prime.

Definition 3.4.7. Let n, a, R and F be as in definition 3.3.6. Let $x \geq 0$ and $y = 2R - xF$ where $0 \leq y < F$.
If $F \geq \sqrt[3]{n}$ and $y^2 - 4x \neq 0$ and not perfect square, n is prime.

The proofs for these definitions can be found in [**?**].

Trial Division is required for Maurer's algorithm. Trial division is an integer factorisation algorithm, but the process can be used for finding primes. For small numbers, e.g. $n < 20$, then trial division is faster than most methods. However, for primes larger than 20-25 bits it is excessively inefficient[8]. Given a boundary n to check for a prime p, p is divided by all primes less than n. p is not prime if it can be divided by any prime in the interval $[1, n]$.

```java
public class TrialDivisionTest {
  public int checkPrime(int n) {
    // If even..
    if (n%2 == 0) return 2;

    for (int x=3;x<(int)Math.sqrt((double)n);x+=2) {
      if (n%x == 0) return x;
    }

    return -1;
  }
}
```

This code could be made more efficient by introducing the use of a prime sieve to pregenerate a list of prime divisors to be used.

Greatest Common Divisor, or GCD, is used throughout our algorithms so far. Having been defined previously, we know what GCD does but not how it would be implemented in Java.

One of the faster, and simpler, algorithms is Euclid's Algorithm. Given two integers p and q, $\gcd(p, q)$=$\gcd(q, p \mod q)$ until $q = 0$.

A recursive algorithm of this is perhaps not the easiest to understand, but certainly the most elegant solution:

```java
public static int gcd(int p, int q) {
  if (q == 0) return p;
  else return gcd(q, p % q);
}
```

We will not be implementing Maurer's algorithm in Java due to the complexity required for little benefit. See §3.3.1.10.

3.4.1.9 BigInteger Generation

Using the constructor `BigInteger(int bitLength, int certainty, Random rnd)`, we can generate random, arbitrary probable primes with the given certainty. The `BigInteger` prime generator uses the Miller–Rabin test[9] with a random number generator to produce random probable primes.

3.4.1.10 Primality Test Outcomes

Given these algorithms, which is best suited to our use further in this project? It is best to attempt to do this formally, rather than subjectively.

Primality *proving* algorithms are required to satisfy the following four constraints, however they also apply to probable primality tests:

1. **Generality** – The primality of any general number can be identified.
2. **Polynomial** – The maximum running time of the algorithm is polynomial.
3. **Deterministic** – The algorithm can deterministically identify if the input integer is prime or not.
4. **Unconditional** – The validity of the test result is not dependent on the proof of a current unproven hypothesis.

Algorithm	**Constraints matched**
Fermat's Little Theorem	Meets only two constraints.
Miller–Rabin	Miller–Rabin in its suggested form meets three constraints, as it is dependent on the proving of the generalised Riemann hypothesis. Basic Miller–Rabin tests are not deterministic, and only meet two constraints.
Solovay–Strassen	Meets all but the third constraint, as it is not deterministic, but probabilistic.
AKS	Meets all four constraints.

Running Times

Algorithm	**Running Time**
Fermat's Little Theorem	$O(k \times log^2 n \times log\ log\ n \times log\ log\ log\ n)$
Miller–Rabin	$O(k\ log^3 n)$
Solovay–Strassen	$O(k\ log^3 n)$
AKS^4	$\breve{O}(log^{12} n)$

The running time of the Maurer algorithm is harder to quantify, due to the number of estimates that have to be made based on long integer mathematics and probability. However, we can use the facts we defined previously to determine a rough estimate of a basic implementation. Where k is the size of the prime needed, and given that we are exponentiating through multiplication and the inverse modulus, for which the running time would be $O(k^3)$, and also dividing, for which the running time would be $O(k \log(k))$, we can say that the running time is $O(\frac{k^4}{log(k)})$[**?**].

Based on the above and what we've seen when implementing the algorithms, we can say with confidence that the Miller-Rabin test is the most useful probable primality test due to the balance of complexity and simplicity in implementation, whereas the AKS test is the most useful provable primality test due to its relatively fast execution speed (compared to other provable prime algorithms), and thus the Miller–Rabin test is the best overall algorithm to be used in the efficient generation of primes.

Conclusion Overall, `BigInteger` gives the best results for minimal execution time, minimal effort, support and reduced likelihood of mistakes. Importantly, it implements the Miller–Rabin test, which as we have determined, is the fastest primality test that can be reasonably used in software. Alongside this, as it is a maintained library that has been in the JDK since 1999, it is mature and any errors or bugs will have been repaired at this point. It is said that a developer should not repeat themselves, or "reinvent the wheel": "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"[10] and as such, reimplementing Miller–Rabin for no reason other than as proof that you can will result in poor software that is difficult and time-consuming to maintain, as well as containing errors.

3.4.1.11 A Note on Random Number Generators

The random number generators mentioned thus far, and all used later, cannot be considered *truly* random. They are known as *pseudo-random number generators* – sequences generated deterministically based on a seed value. The sequences are produced algorithmically based on a relatively small set of values, Generally these sequence can be considered *practically* random for most uses, assuming the seed is not publicly known and is truly random.

Most truly random numbers are generated by analysing physical methods, such as nuclear decay or cosmic background radiation. However, typically this tends to be costly and is usually reserved only for applications that explicitly need high-security, non-deterministic random numbers. Recently the technology has become more available due to the widespread availability of online services – for example, RANDOM.ORG[5] offers an API that returns truly random numbers on request. This naturally comes with its own security downsides, but can be used for non-cryptographic applications. Even then, RANDOM.ORG charges for clients that go over a quota of requests.

The generation and application of random numbers is an extensive and current area of research. For the purposes of the cryptographic utilities to be developed, and the Enigma application, we will be using the *Java SDK* class `SecureRandom`. `SecureRandom` is a cryptographically strong pseudo-random number generator that complies and follows tests specified by [11], section 4.9.1, and also uses non-deterministic seed material, as defined by *RFC 1750*[12]. Assuming the proper use of this class, it is about as close as we will get to generating secure, non-deterministic random numbers without resorting to physical methods.

In short, and for our purposes, the generation of secure random numbers is based around maintaining the secrecy of a truly random seed on which a random sequence can be based.

3.4.2 Strong Primes

A prime n is defined as strong if the integers a, b, and c exist such that:

1. $n - 1$ has a large prime factor a.
2. $n + 1$ has a large prime factor b.
3. $a - 1$ has a large prime factor c.

Gordon's algorithm can be used to generate such strong primes based around the output of a probabilistic primality test such as Miller–Rabin, however it is out of the scope of this section.

The utility of random primes is in the generation of moduli for public-key cryptography. It is expected that the integer primes p, q used to generate modulus n be distinct and of sufficient size that factorisation of n is implausible. However, they should be random in the sense that they are selected from a set of prime integers sufficiently large enough to make a brute force attack infeasible.

Recently it has been shown that strong primes do not make much difference in terms of security over their random prime counterparts. However, it is of shared opinion that as they are no *less* secure and have a negligible execution time, strong primes should still be used for cryptographic prime requirements.

For more information, see [13] or [14].

[5] http://random.org

3.5 Integers

Integers are a well-defined concept, and you would have had great difficulty getting this far in the report without understanding what one is. Integers are, as with lots of areas of studies, extremely useful in cryptography, or perhaps more specifically: the properties of integers and the operations we can perform on them are particularly useful in cryptography.

3.5.1 Factorisation

The integer factorisation problem is the very foundation of many cryptographic algorithms. Formally, it is defined as:

Definition 3.5.1. Given a positive integer n, find its prime factors: $n = p_1^{e_1} p_2^{e_2} ... p_n^{e_n}$ where p_n are distinct primes, and $e_n \geq 1$.

It is this problem that makes it implausible for RSA and other algorithms to be broken by factoring the modulus and obtaining the private keys. We will see this in the following sections.

3.6 Public-key Cryptography

3.6.1 RSA

So far we have only covered the concepts of number theory and how they're used in cryptography. In this section we will discuss exactly how they are implemented, and go in to greater detail about the algorithms and techniques required. §2 provides a brief definition of public-key cryptography.

3.6.1.1 The RSA Problem

The RSA Problem forms the foundation of the RSA public key system:

Definition 3.6.1. Given 3 parameters:

1. a positive integer n, $n = pq$ where p and q are distinct, odd primes.
2. a positive integer e, where $\gcd(e, (p-1)(q-1)) = 1$.
3. an integer c.

Find integer m, where $m^e = c \mod n$.

3.6.1.2 RSA

The RSA cryptosystem – an implementation of the RSA problem – is the most popular asymmetric cryptosystem, mostly due to its ease of implementation and the simple concepts behind it. Its primary purpose is to provide privacy and confidentiality, and we will be using it as part of a key agreement scheme for use in symmetric ciphers.

3.6.1.3 Key Generation

A fundamental property of a public-key cryptosystem is the ability to generate and handle public- and private-keys, both of which come in pairs of corresponding keys. They complete the encryption and decryption transformations, respectively.

When wishing to setup the ability to receive asymmetrically encrypted messages, a user must first generate their keys as so:

1. Generate two large primes, p and q, and calculate $n = pq$.
2. Calculate $\phi = (p-1)(q-1)$, and randomly pick an integer e until $1 < e < \phi$, assuming $\gcd(e, \phi) = 1$.
3. Calculate integer d, $1 < d < \phi$ so that $ed \equiv 1(\mod \phi)$

The public key is (n, e) and the private key is d.

The Java implementation of this as a class is:

```java
package com.cyanoryx.uni.crypto.rsa;

import java.math.BigInteger;
import java.security.SecureRandom;

/**
 *
 *
 * @author adammulligan
 *
 */
public class KeyGenerator {
  public static int SIZE_DEFAULT = 1024, SIZE_MAX = 16384, SIZE_MIN = 256;

  private int keysize = SIZE_DEFAULT;

  /**
   * Constructor
   *
   * @param size Size of the keys in bits
   */
  public KeyGenerator(int size) throws InternalError {
    if (size<SIZE_MAX) {
      this.keysize = size;
    } else {
      throw new InternalError("Key size must adhere to "
                                     +SIZE_MIN+" < k < "+SIZE_MAX);
    }
  }

  public BigInteger[] generatePair() {
    BigInteger[] pq = this.generatePrimes(80);

    BigInteger p = pq[0];
    BigInteger q = pq[1];

    // N = pq
    BigInteger N = p.multiply(q);

    // r = (p-1)*(q-1)
    BigInteger phi = p.subtract(BigInteger.valueOf(1));
    phi = phi.multiply(q.subtract(BigInteger.valueOf(1)));

    BigInteger E;

    do {
      E = new BigInteger(this.keysize/2,
                         new SecureRandom());
    } while((E.compareTo(phi)!=-1)
            || (E.gcd(phi)
            .compareTo(BigInteger.valueOf(1))!=0));

    // D = 1/E mod r
    BigInteger D = E.modInverse(phi);

    return new BigInteger[]{N,E,D};
  }

  private BigInteger[] generatePrimes(int certainty) {
    BigInteger p,q;

    p = new BigInteger(this.keysize/2,certainty,
                       new SecureRandom());
    q = new BigInteger(this.keysize/2,certainty,
                       new SecureRandom());

    if (p == q) this.generatePrimes(certainty);

    BigInteger[] pq = new BigInteger[2];
    pq[0] = p;
    pq[1] = q;

    return pq;
  }
}
```

This file can be found at *src/com/cyanoryx/uni/crypto/rsa/KeyGenerator.java*.

Key Distribution Public-keys, as can be determined from their name, are intended to be made public and widely available. This has its own problems however, as without any capability to verify the origin of a public-key, a user will be susceptible to impersonation attacks. There are various methods to solve this, including certificates, using a trusted server, and other techniques that will be discussed in §5.

3.6.1.4 Encryption

For a user Alice to send an encrypted message m to user Bob, Alice must obtain Bob's public key (n, e). The message is encrypted, $c = m^e \mod n$, and can be sent securely to Bob.

The Java implementation of encryption is:

```java
package com.adammulligan.uni;

```

```java
/**
 *
 * @author adammulligan
 *
 */
public class Encrypt {
  private String key_file;

  private int length;

  private BigInteger E,N;
  private BigInteger[] ciphertext;

  /**
   *
   * @param key_file The public key file for encryption
   */
  public Encrypt(String key_file) {
    this.key_file = key_file;
  }

  /**
   * Takes a String, encrypts using RSA Basic and stores locally
   *
   * @param message Plaintext to be encrypted
   */
  private void encrypt(String message) {
    byte[] temp = new byte[1], bytes;

    bytes = message.getBytes();

    BigInteger[] converted_bytes = new BigInteger[bytes.length];

    // Convert each byte of the message into a bigint
    for(int i=0; i<converted_bytes.length;i++) {
      temp[0] = bytes[i];
      converted_bytes[i] = new BigInteger(temp);
    }

    this.ciphertext = new BigInteger[converted_bytes.length];

    // The actual encryption!
    // Loop through the array of bigint bytes m,
    // and create an array making c = m
    for(int i=0;i<converted_bytes.length;i++) {
      this.ciphertext[i] = converted_bytes[i].modPow(this.E,
                                                     this.N);
    }
  }

  private void log(String msg) {
    System.out.println(msg);
  }
}
```

3.6.1.5 Decryption

Decryption, as with encryption, is simple: using private key d, Bob computes $m = c^d \bmod n$.

The Java implementation of decryption is:

```java
package com.adammulligan.uni;


/**
 * An interactive console application that encrypts
 * a file based on a provided RSA key pair.
 *
 * @author adammulligan
 *
 */
public class Decrypt {
  private String key_file;

  private BigInteger E,N;
  private BigInteger[] plaintext;

  /**
   *
   * @param key_file The public key file for encryption
   */
  public Decrypt(String key_file,String input_file,String output_file) {
    this.key_file = key_file;
  }

  /**
   * Takes a String, encrypts using RSA Basic and stores locally
   *
   * @param message Ciphertext to be decrypted
   */
  private void decrypt(String message) {
    byte[] temp = new byte[1], bytes;

    bytes = message.getBytes();

    BigInteger[] converted_bytes = new BigInteger[bytes.length];

    // Convert each byte of the message into a bigint
    for(int i=0; i<converted_bytes.length;i++) {
      temp[0] = bytes[i];
      converted_bytes[i] = new BigInteger(temp);
    }

    this.plaintext = new BigInteger[converted_bytes.length];

    // The actual ecryption!
    // Loop through the array of bigint bytes m,
    // and create an array making c = m
    for(int i=0;i<converted_bytes.length;i++) {
      this.plaintext[i] = converted_bytes[i].modPow(this.E, this.N);
    }
  }

  private void log(String msg) {
    System.out.println(msg);
  }
}
```

Both the encryption and decryption Java files have some helper methods removed for brevity and to highlight the actual algorithm.

3.6.1.6 Formal Proof

Given any message m such that $m \in \mathbb{Z}_n$, as the decryption transformation $D(m)$ is an inverse of the encryption transformation $E(m)$, we have:

$$E(D(m)) = D(E(m)) = m^{ed} \mod n$$

Exponents e and d are multiplicative inverses of each other, modulo $\phi(n)$, we get:

$$ed = 1 + k(p-1)(q-1)$$

$$\begin{array}{rcll} m^{ed} & \equiv & m(m^{p-1})^{k(q-1)} & (\mod p) \\ & \equiv & m((m \mod p)^{p-1})^{k(q-1)} & (\mod p) \\ & \equiv & m(1)^{k(q-1)} & (\mod p) \\ & \equiv & m & (\mod p) \end{array}$$

And thus $m^{ed} \equiv m \mod p$.

3.6.1.7 Textbooks and OAEP

So far we have looked at what are known as “textbook algorithms” – algorithms that are not suited for use in the real world. Generally they are constructed around the core mathematics of the problem, without any additional measures to counteract basic attacks, like frequency analysis.

As a solution to this, a number of standards have been introduced that add extra requirements to the algorithms such as hashing to improve the overall security. In the case of RSA, the most common standard implemented is known as *RSA OAEP* – RSA Optimal Asymmetric Encryption Padding[15]. It is based upon the EME-OAEP encoding method – the purpose of this scheme is to add randomness to the encrypted output, creating a probabilistic result, along with preventing any information leakage from ciphertexts.

More specifically, the use of OAEP protects against chosen-plaintext attacks. There are some concerns about the inability to prove that it is secure from certain types of chosen-plaintext attacks, however this is discussed in more detail in §7.

Given:

- A hash function.
- A mask generation function (MGF)
- A public key.
- The plaintext message.
- A label to be appended to the message.

There are three steps to the RSA OAEP encryption process:

1. Check the length of input to ensure that it is below the limit for the hash function. In our case, the limit is $2^{61} - 1$ (SHA-1).
2. Encode the message using EME-OAEP:
3. (a) Construct the `byte` array:

   ```
   DB = LHash (the hash of the label)
       || PS (a byte array of zero-bytes [0x00])
       || 0x01
       || M (the message to encrypt)
   ```

   (b) Generate a random byte array for us as a seed s.

   (c) Create a mask $m =$ MGF(s, rsa modulus length $-$ hash length -1)

   (d) Mask DB: $mdb = DB \oplus m$

   (e) Create a seed mask $sm =$ MGF(masked DB, hash length)

   (f) Mask the seed: $ms = s \oplus sm$

   (g) Finally, concatenate the generated masked byte arrays, prefixed with `0x00`: `EM = 0x00 || ms || mdb`.
4. Encrypt the encoded `byte` array EM using the RSA method as defined previously.

After all the components have been created, they are assembled as so[16]:

```
                    +----------+---------+-------+
               DB = |  lHash   |    PS   |   M   |
                    +----------+---------+-------+
                                   |
         +----------+              V
         |   seed   |--> MGF ---> xor
         +----------+              |
               |                   |
      +--+     V                   |
      |00|    xor <----- MGF <-----|
      +--+     |                   |
        |      |                   |
        V      V                   V
      +--+----------+----------------------------+
EM =  |00|maskedSeed|          maskedDB          |
      +--+----------+----------------------------+
```

However, there is one function with this that is not yet explained.

The Mask Generation Function (MGF) is a function that takes (in Java terminology) a `byte` array of arbitrary length and a desire output length, and gives a `byte` array of that length. As it is based on a hash function, the output is deterministic. It is defined the RSA PKCSv2 specification[16] that the MGF should be pseudorandom, in that given one piece of the output, you are unable to predict another part of the output. The entire security of RSA OAEP is dependent on this property.

The MGF in Java is:

```
package com.cyanoryx.uni.crypto.rsa;

import java.security.MessageDigest;

import com.cyanoryx.uni.common.Bytes;

/**
 *  A mask generation function takes an octet string of variable length
 *  and a desired output length as input, and outputs an octet string of
 *  the desired length. There may be restrictions on the length of the
 *  input and output octet strings, but such bounds are generally very
 *  large.  Mask generation functions are deterministic; the octet string
 *  output is completely determined by the input octet string.
 *
 * @author adammulligan
 *
 */
public class MGF1 {
  private final MessageDigest digest;

  public MGF1(MessageDigest digest) {
    this.digest = digest;
  }


  /**
   *  MGF1 is a Mask Generation Function based on a hash function.
   *
   *  MGF1 (mgfSeed, maskLen)
   *
   *  Options:
   *  Hash     hash function (hLen denotes the length in octets of the hash
   *           function output)
   *
   *  Input:
   *  mgfSeed  seed from which mask is generated, an octet string
   *  maskLen  intended length in octets of the mask, at most 2^32 hLen
   *
   *  Output:
   *  mask     mask, an octet string of length maskLen
   *
   * @param mgfSeed
   * @param maskLen
   * @return
   */
  public byte[] generateMask(byte[] mgfSeed, int maskLen) {
  // (maskLen / hLen) - 1
    int hashCount = (maskLen + this.digest.getDigestLength() - 1)
                / this.digest.getDigestLength();

    byte[] mask = new byte[0];

    // For counter from 0 to \ceil (maskLen / hLen) - 1, do the following:
    for (int i=0;i<hashCount;i++) {
      /*
       * a. Convert counter to an octet string C of length 4 octets (see
       *    Section 4.1):
```

```
       *
       *     C = I2OSP (counter, 4) .
       */
      this.digest.update(mgfSeed);
      this.digest.update(new byte[3]);
      this.digest.update((byte)i);
      byte[] hash = this.digest.digest();

      /*
       * b. Concatenate the hash of the seed mgfSeed and C to the octet
       *    string T:
       *
       *     T = T || Hash(mgfSeed || C)
       */
      mask = Bytes.concat(mask, hash);
    }

    //  4. Output the leading maskLen octets of T as the octet string mask.
    byte[] output = new byte[maskLen];
    System.arraycopy(mask, 0, output, 0, output.length);
    return output;
  }
}
```

This file can be found in *com.cyanoryx.uni.crypto.rsa*.

Decryption, as with standard RSA, is effectively the reverse of encryption. The ciphertext is decrypted, the concatenated string of properties is split in to the components created in encryption, and the message output.

Because OAEP is a purely technical standard, rather than mathematical, we will list the Java implementation of it with inline comments as explanation.

```
package com.cyanoryx.uni.crypto.rsa;

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.zip.DataFormatException;

import com.cyanoryx.uni.common.Bytes;

public class RSA_OAEP {
  private BigInteger E,N;

  private Key key;

  private MessageDigest md;

  public RSA_OAEP(Key key) throws NoSuchAlgorithmException {
    this.key = key;

    this.E = this.key.getExponent();
    this.N = this.key.getN();

    this.md = MessageDigest.getInstance("sha1");
  }

```

```
/**
 * Encrypts a byte[] using RSA-OAEP and returns a byte[]
 * of encrypted values
 *
 * @param byte[] M byte array to be encrypted
 * @return byte[] C byte array of encrypted values
 * @throws DataFormatException
 */
public byte[] encrypt(byte[] M) throws DataFormatException,
                                       InternalError {
  if (this.key instanceof PrivateKey) {
    throw new InternalError("A public key must be " +
        "passed for encryption");
  }

  // The comments documenting this function are from
  // RFC 2437 PKCS#1 v2.0

  /*
     1. Length checking:

        a. If the length of L is greater than the input
           limitation for the hash function
           (2^61 - 1 octets for SHA-1), output
           "label too long" and stop.

        b. If mLen > k - 2hLen - 2, output "message too long"
           and stop.
   */
  int mLen = M.length;

  int k = (this.N.bitLength()+7)/8;

  if (mLen > (k - 2*this.md.getDigestLength() - 2)) {
    throw new DataFormatException("Block size too large");
  }

  /*
     b. Generate an octet string PS consisting of k - mLen - 2hLen - 2
         zero octets.  The length of PS may be zero.
   */
  byte[] PS = new byte[k - mLen - 2*this.md.getDigestLength() - 2];

  /*
      c. Concatenate lHash, PS, a single octet with hexadecimal value
         0x01, and the message M to form a data block DB of length k -
         hLen - 1 octets as
              DB = lHash || PS || 0x01 || M.
   */
  byte[] DB = Bytes.concat(this.md.digest(),PS, new byte[]{0x01},M);

  // d. Generate a random octet string seed of length hLen
  // (hLen = hash function output length in octets)
  SecureRandom rng = new SecureRandom();
  byte[] seed = new byte[this.md.getDigestLength()];
  rng.nextBytes(seed);

  // e. Let dbMask = MGF(seed, k - hLen - 1).
  MGF1 mgf1 = new MGF1(this.md);
  byte[] dbMask = mgf1.generateMask(seed,
      k - this.md.getDigestLength() - 1);

```

```java
    // f. Let maskedDB = DB \xor dbMask.
    byte[] maskedDB = Bytes.xor(DB, dbMask);

    // g. Let seedMask = MGF(maskedDB, hLen).
    byte[] seedMask = mgf1.generateMask(maskedDB,
        this.md.getDigestLength());

    // h. Let maskedSeed = seed \xor seedMask.
    byte[] maskedSeed = Bytes.xor(seed, seedMask);

    /*
       i. Concatenate a single octet with hexadecimal value 0x00,
           maskedSeed, and maskedDB to form an encoded message EM of
           length k octets as

                EM = 0x00 || maskedSeed || maskedDB.
     */
    byte[] EM = Bytes.concat(new byte[]{ 0x00 }, maskedSeed, maskedDB);

    /*
       a. Convert the encoded message EM to an integer message
              representative m (see Section 4.2):

                m = OS2IP (EM).
     */
    BigInteger m = new BigInteger(1,EM);

    /*
       b. Apply the RSAEP encryption primitive (Section 5.1.1) to the RSA
              public key (n, e) and the message representative m to produce
               an integer ciphertext representative c:

                c = RSAEP ((n, e), m).
     */
    BigInteger c = m.modPow(this.E,this.N);

    /*
       c. Convert the ciphertext representative c to a ciphertext C of
              length k octets (see Section 4.1):

                C = I2OSP (c, k).
     */
    byte[] C = Bytes.toFixedLenByteArray(c, k);

    if(C.length != k) {
      throw new DataFormatException();
    }

    return C;
  }

  /**
   * Takes a byte[] of encrypted values and uses
   * RSAES-OAEP-DECRYPT to decrypt them
   *
   * @param byte[] C Array of encrypted values
   * @return byte[] M Array of decrypted values
   * @throws DataFormatException
   */
  public byte[] decrypt(byte[] C) throws DataFormatException,
                                         InternalError {
    if (this.key instanceof PublicKey) {
      throw new InternalError("A private key must be " +
          "passed for decryption");
    }

    /*
       1. Length checking:

          a. If the length of L is greater than the input
             limitation for the hash function
             (2^61 - 1 octets for SHA-1), output "decryption
             error" and stop. (NOTE: labels are not used in
             this implementation)

          b. If the length of the ciphertext C is not k
             octets, output "decryption error" and stop.
     */
    int k = (this.N.bitLength()+7)/8;

    if(C.length != k) {
      throw new DataFormatException();
    }

    //c. If k < 2hLen + 2, output "decryption error" and stop

    if (k < (2*this.md.getDigestLength()+2)) {
      throw new DataFormatException("Decryption error");
    }

    /*
       2.    RSA decryption:

            a. Convert the ciphertext C to an integer ciphertext
               representative c (see Section 4.2):

                  c = OS2IP (C).
     */
    BigInteger c = new BigInteger(1, C);

    /*
       b. Apply the RSADP decryption primitive (Section 5.1.2) to the
           RSA private key K and the ciphertext representative c to
           produce an integer message representative m:

                m = RSADP (K, c).
     */
    BigInteger m = c.modPow(this.E,this.N);

    /*
       c. Convert the message representative m to an encoded message EM
           of length k octets (see Section 4.1):

                EM = I2OSP (m, k).
     */
    byte[] EM = Bytes.toFixedLenByteArray(m, k);
    if(EM.length != k) {
        throw new DataFormatException();
    }

    /*
       3. EME-OAEP decoding:

          a. If the label L is not provided, let L be the empty string.
```

```
         Let lHash = Hash(L), an octet string of length hLen
         (see the note in Section 7.1.1).

      b. Separate the encoded message EM into a single octet Y,
         an octet string maskedSeed of length hLen, and an octet
         string maskedDB of length k - hLen - 1 as

            EM = Y || maskedSeed || maskedDB.
 */
if(EM[0] != 0x00) throw new DataFormatException();

byte[] maskedSeed = new byte[this.md.getDigestLength()];
System.arraycopy(EM, 1, maskedSeed, 0, maskedSeed.length);

byte[] maskedDB = new byte[k - this.md.getDigestLength() -1];
System.arraycopy(EM,
             1 + this.md.getDigestLength(),
             maskedDB, 0, maskedDB.length);

//  c. Let seedMask = MGF (maskedDB, hLen).
MGF1 mgf1 = new MGF1(this.md);
byte[] seedMask = mgf1.generateMask(maskedDB,
                                this.md.getDigestLength());

//  d. Let seed = maskedSeed ^ seedMask.
byte[] seed = Bytes.xor(maskedSeed, seedMask);

//  e. Let dbMask = MGF (seed, k - hLen - 1).
byte[] dbMask = mgf1.generateMask(seed,
                              k - this.md.getDigestLength() -1);

//  f. Let DB = maskedDB ^ dbMask.
byte[] DB = Bytes.xor(maskedDB, dbMask);

/*
   g. Separate DB into an octet string lHash' of length hLen, a
       (possibly empty) padding string PS consisting of octets with
       hexadecimal value 0x00, and a message M as

            DB = lHash' || PS || 0x01 || M.
 */
byte[] lHash1 = new byte[this.md.getDigestLength()];
System.arraycopy(DB, 0, lHash1, 0, lHash1.length);
if(!Bytes.equals(this.md.digest(), lHash1))
    throw new DataFormatException("Decryption error");

/*
   If there is no octet with hexadecimal value 0x01 to separate PS
    from M, if lHash does not equal lHash', or if Y is nonzero,
    output "decryption error" and stop.  (See the note below.)
 */
int i;
for(i = this.md.getDigestLength(); i < DB.length; i++) {
    if(DB[i] != 0x00) break;
}

if(DB[i++] != 0x01) throw new DataFormatException();

// 4. Output the message M.
int mLen = DB.length - i;
byte[] M = new byte[mLen];
System.arraycopy(DB, i, M, 0, mLen);
```

```
      return M;
   }
}
```

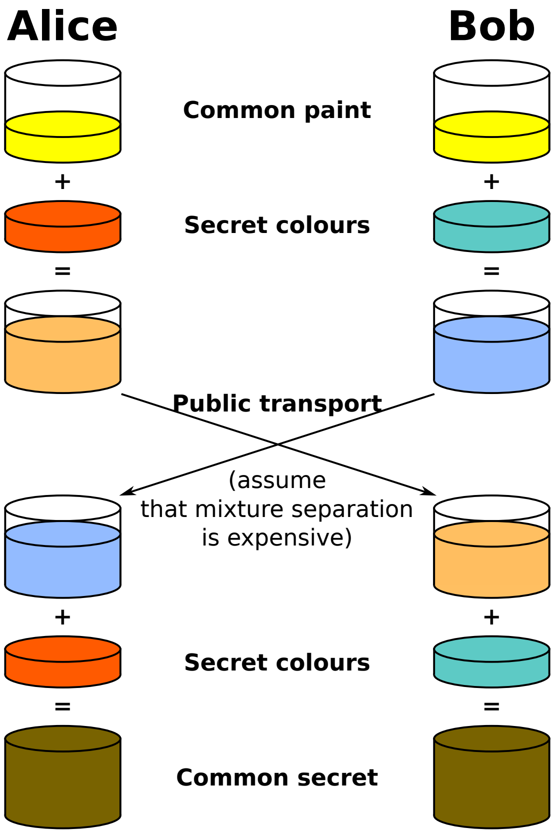
The Key class is a method for objectively storing key information that is imported from file. It can be viewed in the same package, *com.cyanoryx.uni.crypto.rsa*.

3.6.2 Diffie–Hellman

The Diffie-Hellman Key Exchange Protocol was one of the first public-key schemes created for sharing keys. Created by Whitfield Diffie, and Martin Hellman (though believed to have been originally created by British Signals Intelligence agency *GCHQ* and kept classified), it allows to users to generate keying material without ever having to send the key in cleartext.

The general concept of the D–H protocol is that a large prime and a generator element are shared publicly between entities, both of whom will then pick a secret number and generate a public number based upon this. Each entity uses their partner's public number to generate keying material, which will be the same for both.

There is an excellent public domain graphic that explains the Diffie–Hellman process using the colour of paint[6]:



[6] Created and released in to the public domain by Wikimedia user *Flugaal*

Unfortunately, however, we cannot easily turn graphics in to algorithms, and so we must define the formal process behind Diffie–Hellman[17].

1. Given a large prime p and g, an element in the finite field F_p^*,
2. Alice picks an integer a in $[1, p-1)$ and calculates $g_a = g^a \mod p$ and sends it to Bob.
3. Bob picks an integer b in $[1, p-1)$ and calculates $g_b = g^b \mod p$ and sends it to Alice.
4. Alice and Bob computer $K = g_b^a$ and $K = g_a^b$, respectively.

Given that the relationship $ab \equiv ba \mod (p-1)$ is valid, and the key for Alice is:

$$K = g^{ba} \mod p$$

And for Bob:

$$K = g^{ab} \mod p$$

We can see that both Alice and Bob's calculations will result in the same keying material.

The secrecy of Diffie–Hellman is reliant on a problem known as the Computational Diffie–Hellman Problem.

Definition 3.6.2. Given g, a generator element in the finite field $\mathbb{F}_q^*$, and $g^a, g^b \in F_q^*, 0 < a, b < q$, find g^{ab}.

These values are available to an attacker, however they are not useful without being able to calculate g^{ab} easily.

3.6.2.1 Implementation

In Java, two classes are necessary for a robust D–H implementation. Firstly, a key generator to create the public and private keys for both entitites.

```java
package com.cyanoryx.uni.crypto.dh;

import java.math.BigInteger;
import java.security.SecureRandom;

/**
 * Diffie-Hellman Public/Private Key Pair generator
 *
 * See http://www.ietf.org/rfc/rfc2631.txt
 *
 * @author adammulligan
 *
 */
```

```java
public class KeyGenerator {
  public static int SIZE_DEFAULT = 256,
                    SIZE_MAX = 16384,
                    SIZE_MIN = 256;

  private int keysize = SIZE_DEFAULT;

  private BigInteger g,p;

  /**
   * Constructor
   *
   * @param size Size of the keys in bits
   */
  public KeyGenerator(int size) throws InternalError {
    this.setKeySize(size);
  }

  /**
   * Create a key-pair based upon a received public key.
   *
   * @param size
   * @param key
   */
  public KeyGenerator(int size,PublicKey key) {
    this.setKeySize(size);

    this.g = key.getG();
    this.p = key.getP();
  }

  /**
   * Generates a pair of priv/pub keys for a Diffie-Hellman
   * key exchange
   *
   * See section 2.1.1 RFC 2631
   *
   * @return {x,y} - Private/Public DH Key Pair
   */
  public Key[] generatePair() {
    BigInteger[] pq = this.generatePrimes(80);

    /*
     * p is a large prime
     * q is a large prime
     */
    this.p = (this.p == null
          || this.p.compareTo(BigInteger.ONE)==-1)
             ? pq[0]
             : this.p;
    BigInteger q = pq[1];

    // Private key exponent
    // xa is party a's private key
    BigInteger x;
    do {
      x = new BigInteger(p.bitLength()-1,new SecureRandom());
    } while(x.compareTo(BigInteger.ZERO)!=1 ||
        x.compareTo(p.subtract(BigInteger.ONE))!=-1);

    if (this.g == null || this.g.compareTo(BigInteger.ZERO)==0) {
      /*
```

```
       * h is any integer with 1 < h < p-1 such
       * that h{(p-1)/q} mod p > 1
       */
      BigInteger h;
      do {
        h = new BigInteger(p.bitLength()-1,new SecureRandom());
        // h{(p-1)/q} mod p
        g = h.modPow((p.subtract(BigInteger.ONE)).divide(q),p);
      } while (h.compareTo(BigInteger.ONE)!=1 || // 1 < h
           h.compareTo(p.subtract(BigInteger.ONE))!=-1 || // h < p-1
           g.compareTo(BigInteger.ONE)!=1); // h{(p-1)/q} mod p > 1
    }

    // Public key
    // public key; ya = g ^ xa mod p
    BigInteger y = this.g.modPow(x,p);

    Key priv = new PrivateKey(x);
    Key pub  = new PublicKey(y,this.g,this.p);

    // {Priv,Pub}
    return new Key[]{priv,pub};
  }

  /**
   * Generates a pair of prime numbers using java.math.BigInteger
   *
   * @param certainty - Prime certainty
   * @return {p,q} - Prime pair
   */
  private BigInteger[] generatePrimes(int certainty) {
    BigInteger p,q;

    p = new BigInteger(this.keysize/2,certainty,new SecureRandom());
    q = new BigInteger(this.keysize/2,certainty,new SecureRandom());

    if (p == q) this.generatePrimes(certainty);

    BigInteger[] pq = new BigInteger[2];
    pq[0] = p;
    pq[1] = q;

    return pq;
  }

  private void setKeySize(int size) throws InternalError {
    if (size<SIZE_MAX) {
      this.keysize = size;
    } else {
      throw new InternalError("Key size must adhere " +
          "to "+SIZE_MIN+" < k < "+SIZE_MAX);
    }
  }
}
```

Secondly, the actual algorithm implementation that, given a received public and local private key, will generate keying material as defined in RFC2631[18].

```
package com.cyanoryx.uni.crypto.dh;

```

```java
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

import com.cyanoryx.uni.common.Bytes;

/**
 * Diffie-Hellman Key Agreement protocol
 * Takes a private key, and received public key to generate a shared
 * secret between two parties
 *
 * @author adammulligan
 *
 */
public class KeyAgreement {
  private PublicKey sender_key;

  private MessageDigest md;

  private BigInteger x,counter;

  /**
   *
   * @param priv_key Personal private key
   * @param sender_key Public key of the second party
   */
  public KeyAgreement(PrivateKey priv_key, PublicKey sender_key) {
    this.sender_key = sender_key;

    this.counter = new BigInteger("0");

    try {
      this.md = MessageDigest.getInstance("sha1");
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }

    this.x = priv_key.getX();
  }

  /**
   * Generates ZZ, as defined in RFC 2631
   *
   * @return ZZ - byte array where ZZ = yB^x (mod p)
   */
  private byte[] generateSharedSecret() {
    BigInteger y = this.sender_key.getY();
    BigInteger p = this.sender_key.getP();

    BigInteger ZZ = y.modPow(this.x, p);

    return ZZ.toByteArray();
  }

  /**
   * Generates KEK, as defined in RFC 2631
   *
   * @return KEK - byte array where
   *   KEK = ZZ || (OID || counter || Key Length)
   */
  public byte[] generateKeyMaterial() {
    this.counter = this.counter.add(BigInteger.ONE);

    byte[] ZZ = this.generateSharedSecret();

    byte[] OID = "2.16.840.1.101.3.4.1".getBytes();
    byte[] KeyLength = String.format("%x",
                       new BigInteger("256".getBytes())).getBytes();
    byte[] Counter = Bytes.toFixedLenByteArray(this.counter,4);

    byte[] KEK = Bytes.concat(ZZ,Bytes.concat(OID,Counter,KeyLength));

    this.md.update(KEK);
    return this.md.digest();
  }
}
```

Both `KeyGenerator` and `KeyAgreement` can be found in *com.cyanoryx.uni.crypto.dh.*

Using these implementations, a shared-key would be generated as so:

1. Alice generates a public and private key pair with `KeyGenerator`, and sends the public half to Bob.
2. Bob generates a key pair based on Alice's key using `KeyGenerator`, and sends the public half to Alice.
3. Both Alice and Bob use `KeyAgreement` to generate keying material using both their private and public keys.
4. Alice and Bob are now in possession of a shared session key that has not been sent between them, let alone in cleartext.

3.7 Other Variations of Public-key Cryptography

3.7.1 ElGamal

ElGamal differs from RSA and D–H in that it is not just a public-key encryption/decryption algorithm, but a public-key cryptosystem that is to be used not only to generate and share keys, but also encrypt the confidential messages to be sent using these keys. It utilises the one-way trapdoor function, and became popular due to the fact that it is based around the Computational Diffie–Hellman Problem and the Discrete Logarithm Problem, [17] §pg.253, the latter of which is considered as an alternative to the integer factorisation problem.

Key Generation and Sharing Alice:

1. Chooses a random prime number p.
2. As with D–H, calculates a multiplicative generator element g in $\mathbb{F}_p^*$.
3. Picks a random number d where $d \in \mathbb{Z}_{p-1}$.
4. Computes $e = g^x \mod p$.
5. d and e are the private and public keys, respectively.

Encryption If Bob wishes to send a message m to Alice, he must:

1. Pick a random integer k such that $k \in \mathbb{Z}_{p-1}$.
2. Computer a pair of ciphertexts (C_1, C_2) such that:
3. $C_1 = g^k \mod p$
4. $C_2 = e^k m \mod p$

And send the pair to Alice.

Decryption Upon receipt, Alice:

1. Calculates $m = C_1/C_2^d \mod p$.

This can be proved by showing that:

$$C_1^d \equiv (g^k)^x \equiv (g^d)^k \equiv e^k \equiv C_2/(m \mod p)$$

3.7.1.1 Textbook Algorithm Insecurity

As we have said, textbook algorithms are defined as such because they are simply the core mathematical concept, with no additional security measures, and thus are very weak encryption schemes. The same stands for ElGamal – it can leak parts of information. Commonly, a value g (as defined above) is of order $r = \text{ord}_p(g) << p$ to improve efficiency[17] however if the message m is not defined in the group $\langle g \rangle$ then a meet-in-the-middle attack – for example, calculating the encrypted value of the plaintext for all possible keys and then decrypting for each key, which is likely to revel the correct keys – can be made.

For a ciphertext $(C_1, C_2) = (g^k, e^k m) \mod p$, an attacker Mallory can calculate $C_2^r \equiv m^r \mod p$. This is bad because it results in ElGamal no longer being a probabilistic encryption scheme, but a deterministic one. This means that a trial-and-error method of determining keys can take place, as above.

3.7.2 Rabin

The Rabin Public-Key Algorithm, created by Michael O. Rabin of Miller–Rabin fame, is a public-key cryptosystem that is based around the difficulty of calculating a square root, modulo a non-prime integer, which is equivalent to the difficulty of integer factorisation. It is a simple and efficient algorithm, and is implemented as so:

Key Generation Alice generates a key as in RSA:

1. Generate to large primes, p and q.
2. Compute the modulus $N = pq$.
3. Pick a random integer b such that $b \in \mathbb{Z}_N^*$.
4. (N, b) becomes the public key, while (p, q) is the private key.

Encryption To encrypt a message m, Bob must:

1. Calculate ciphertext $c = m(m + b) \mod N$.

Decryption To decrypt a received ciphertext c, Alice must:

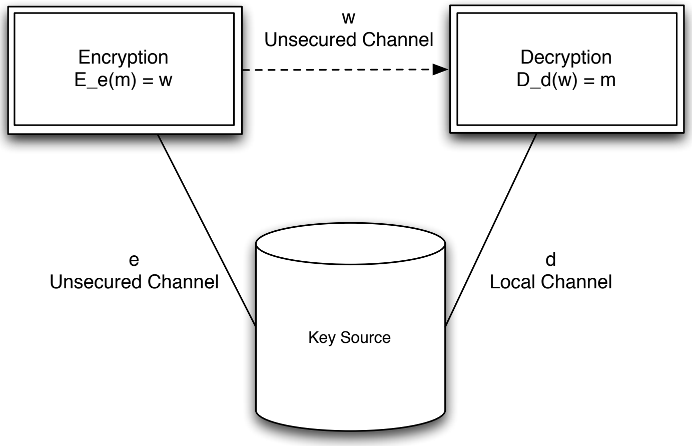
1. Solve the equation $m^2 + bm - c \equiv 0 \mod N$.

Due to its popularity, we will be using RSA as our main method of key exchange.

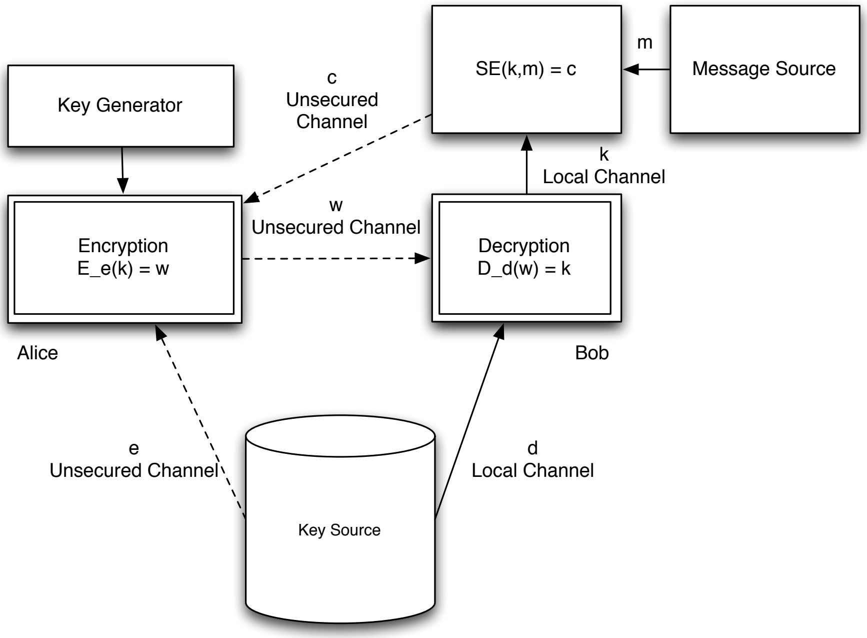
3.8 Uses and going forward

Now that we have formally defined the mathematics behind public-key cryptography and implemented them from prime number generation through to key generation and actual encipherment/decipherment, what can it be used for practically?

In the context of our instant messaging application, could a secure public-key scheme like RSA be used to encrypt outgoing messages? Technically speaking, yes; however, practically, no primarily because of running time concerns: public-key cryptography is significantly slower than using symmetric-key cryptography (see below). The solution to this is to utilise both schemes such that they complement one another – this is known as a digital envelope scheme:



This is the standard process for sending an encrypted message m via public-key cryptography. If instead we encrypt a session key for a symmetric algorithm, like AES, and send that securely then both entities will be in possession of a session key without it ever having been transferred in the clear:



In this case, Alice generates and encrypts a key k with Bob's public key and sends it to him, Bob then decrypts this and stores the key. Both Alice and Bob are now have the session key.

This is a simple and easy to implement solution using existing – and open source – technologies (without requiring any custom hardware) that we will be utilising as a key exchange mechanism in the Enigma application.

3.8.1 Asymmetric and symmetric algorithm comparison

We have claimed that symmetric cryptographic algorithms are faster than asymmetric, but is that proven when creating even the simplest of implementations? Here we have written a short program that can be used to run an asymmetric and symmetric algorithm (using the Java SDK's built in libraries) and timed with the average length of execution calculated.

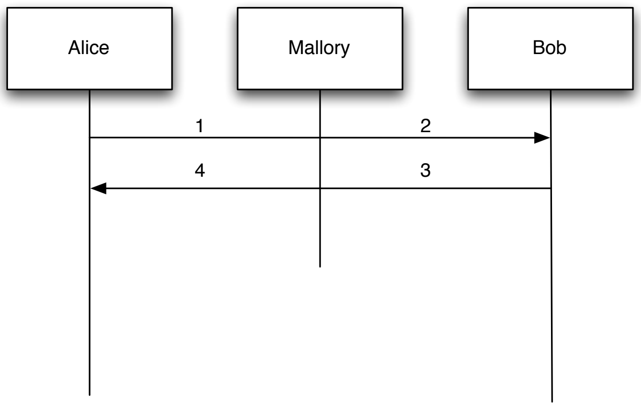
Algorithm	Average Time (20 iterations; nanoseconds)
RSA	14595750
AES	428150
Difference	14167600

As we can see, the symmetric algorithm (AES in this case) is *considerably* faster than the asymmetric for encrypting data.

```
byte[] tmp = new byte[16];
new Random().nextBytes(tmp);

RSA_OAEP rsa = new RSA_OAEP(new PublicKey(new File("./id_rsa")));

final long rsa_startTime = System.nanoTime();
final long rsa_endTime;
try {
  rsa.encrypt(tmp);
} finally {
  rsa_endTime = System.nanoTime();
}
final long rsa_duration = (rsa_endTime - rsa_startTime);

System.out.println("Asymmetric time: "+rsa_duration+"ns");

AES aes = new AES();
aes.setKey(new Key(KeySize.K256));
aes.setMode(Mode.ECB);
aes.setPlainText(tmp);

final long aes_startTime = System.nanoTime();
final long aes_endTime;
try {
  aes.encrypt();
} finally {
  aes_endTime = System.nanoTime();
}
final long aes_duration = aes_endTime - aes_startTime;

System.out.println("Symmetric time:  "+aes_duration+"ns");

System.out.println((rsa_duration > aes_duration)
               ? "AES faster by    "+(rsa_duration-aes_duration)+"ns"
               : "RSA faster by    "+(aes_duration-rsa_duration)+"ns");
```

3.8.2 Problems

As considered in §2.3.5, there is an issue with using public-key schemes (or indeed any basic cryptosystem) for either message encryption or key encryption – impersonation. The ideal public-key system should allow two entities to securely communicate by sharing their public keys without ever meeting to exchange session keys, however there is no “built-in” method for determining whether or not the public key you are using to encrypt a key is indeed the key of the entity with which you wish to talk to.



1. Alice sends her public key to who she thinks is Bob, however Mallory receives it.
2. Mallory sends her own public key to Bob, masquerading as Alice.
3. Bob encrypts a generated session key with Mallory's public key, and sends it to Mallory.
4. Mallory decrypts the session key, and re-encrypts it with Alice's public key, and sends it to Alice.

Here, Mallory is acting as a middle-man and unbeknownst to the other she gains access to the session key. Using this, she will be able to read and modify any messages sent between Alice and Bob, undetected.

§2.2 states that there are four requirements are necessary for a system to be considered secure:

1. Confidentiality
2. **Authentication**
3. Non-repudiation
4. Data Integrity

Here we are failing **authentication**. As we will see in §5, we can use another aspect of public-key cryptography – digital signatures – along with trusted third parties to identify and authenticate the entity with which we are communicating.

However, we are getting ahead of ourselves. The next step is to research and implement the symmetric ciphers that will be used to rapidly and securely encrypt the messages that will be transmitted between users.

Chapter 4

Symmetric Cryptography

4.1 Introduction

Symmetric cryptography is a type of *secret-key cryptosystem*, meaning that the encryption and decryption transformations use the same key and the encryption function is one-to-one (and thus invertible). More specifically we can define a symmetric algorithm as a cryptosystem with keys $k_{encrypt}$ and $k_{decrypt}$, where $k_{encrypt} = k_{decrypt}$. This is known as having a shared secret.

Simplified, encryption can be written as:

$$c = e_k(m)$$

Where c is the ciphertext, e the cipher function, k the key, and m the plaintext message. Decryption, this being a *symmetric* cipher, is almost exactly the same:

$$m = e_k(c)$$

Symmetric key algorithms can be implemented in two forms: stream and block ciphers, however we will only be looking at block ciphers.

1. **Stream Ciphers** encrypt and decrypt data one character at a time.
2. **Block Ciphers** differ in that they work on fixed-length groups of data using a transformation.

4.1.1 Differences and Which To Use

We are considering here fundamental properties of *basic* block and stream ciphers.

4.1.1.1 Block Ciphers

A block cipher is a function that maps n-bit blocks of plaintext to n-bit blocks of ciphertext using a transformation function.

Pros	Cons
It is impossible to insert or modify characters within a block without detection.	Single characters of messages are not able to be encrypted (unless the message comprises solely of one character) without the whole block having been received.
Frequency analysis impossible.	Error propagation is high (at least, relative to stream ciphers) as one error can affect a whole block.

4.1.1.2 Stream Ciphers

A stream cipher maps a single character n to an encrypted character n'.

Pros	Cons
Low error propagation.	Low diffusion – frequency analysis possible.
	New messages can be constructed by using parts of older messages.

4.1.1.3 Summary

Overall, what can we say? The matter is mostly a balance of preference and perceived security. In this case, we will be using a block cipher known as the AES (Advanced Encryption Standard) algorithm, for a number of reasons:

1. There are few modern stream ciphers that are well documented compared to AES[1]
2. It is the current standard for encryption as defined by the United States National Institute of Standards and Technology, and is recommended by the United States National Security Agency for securing top secret information.
3. And perhaps most importantly, it is vastly more interesting to implement.

In reality, block ciphers and stream ciphers offer no security benefit over one another, except in some very specific situations.

[1] http://www.ecrypt.eu.org/stream/ was an initiative to find modern day stream ciphers for widespread use

4.2 AES

4.2.1 Overview

AES itself is the name of the *standard* whereas the actual name of the algorithm is known as *Rijndael*, though we will refer to it as the AES algorithm. It is a symmetric block cipher with a block size of 128 bits and a key size of 128, 192, or 256 bits.

4.2.2 Mathematical Preliminaries

AES mostly utilises basic bitwise arithmetic on fixed-size blocks, and so the mathematics behind it is not particularly complex. An understanding of bitwise operators like XOR, bit shifts, and logical operators is required, but all other concepts are explained. However, there is one concept that can be difficult to understand relative to the algorithm without explanation: finite fields.

4.2.2.1 Finite Fields

Finite fields, like most things useful to cryptography, are an important part of number theory. We have used them previously in the Diffie-Hellman protocol, however they were left with little explanation.

Finite fields are, perhaps unsurprisingly, a type of field with a finite number of elements and are a subclass of two other algebraic structures:

1. A field is a ring of elements that are non-zero and are formed under multiplication.
2. A ring is a set of elements with only two operations (addition and multiplication), that must follow a set of properties:
   (a) It is an abelian group under addition.
   (b) Under multiplication, it satisfies the closure, associativity and identity axioms.
   (c) Elements are commutative.
   (d) Elements satisfy the distribution axiom: $a \times (b + c) = a \times b + a \times c$

How is this useful to AES? Only two operations are required for our transformations of bytes – addition (XOR) and multiplication – and we want to limit our calculations to a finite number of possible elements (2^8), both of which a finite field offers. As we will explain later, this also means we can use an irreducible polynomial that limits calculations to be within one byte.

4.2.3 Algorithm

Officially the Rijndael algorithm is a cipher with multiple block and key sizes, however we will only be following the AES standard of a fixed block size (16 bytes) and three key sizes (128,192 and 256 bits).

Given a 16-byte (128 bit) message, which is equal to one block, we have bytes b such that:

$$\begin{pmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{pmatrix}$$

The AES algorithm, like most modern symmetric block ciphers, consists of a number of "rounds" in which the block of data is transformed. We define this round transformation as `Round(State,RoundKey)`, where State is the current 16 byte block selected out of the overall text and the RoundKey is a key derived from the input key using a key schedule as defined in §4.2.3.2. The number of rounds is dependent on the key size:

	Key Length (bits)	Number of Rounds
AES-128	128	10
AES-192	192	12
AES-256	256	14

The `Round` function consists of four functions:

```
Round(State,RoundKey) {
  SubBytes(State)
  ShiftRows(State)
  MixColumns(State)
  AddRoundKey(State,RoundKey)
}
```

It should be noted that the final round, irrelevant of key size, is different in that it does not compute a `MixColumns` transformation.

4.2.3.1 Transformations

The four internal functions within AES are known as *transformations* because they modify the current block of data in some way. Each function is invertible, and as such we will only be describing the transformations for *encryption* – decryption is simply running the round in reverse.

All functions work within a finite field. Addition is performed in GF(2), which presents an easy implementation as bytes are represented in base-2 and so two bytes can be added together using XOR[19]:

$$\texttt{01010111} \oplus \texttt{10000011} \equiv \texttt{11010100}$$

However, multiplication is more complicated. Given two elements within this field, each can have powers of n^7 meaning multiplication of the two will result in n^{14} which is a value outside of the field (thus meaning it cannot be represented within a byte). To handle this, all polynomial multiplications are calculated modulo an irreducible polynomial $f(x)$ over the field GF(2^8):

$$f(x) = x^8 + x^4 + x^3 + x + 1$$

SubBytes transforms the state block by replacing each byte value with a corresponding value in a substitution table known as an *S-Box*. The S-Box is calculated by[20]:

1. Taking the multiplicative inverse in the finite field GF(2^8).
2. Apply an affine transformation:

   $$b_i = b_i \oplus b_{(i+4) \mod 8} \oplus b_{(i+5) \mod 8} \oplus b_{(i+6) \mod 8} \oplus b_{(i+7) \mod 8} \oplus c_i$$

   where b_i is the i^{th} bit of the byte, and c_i is the i^{th} bit of the byte `63`.

Given the S-Box table consisting of 16 rows and 16 columns (0-9a-f), and a byte b, we determine the substitution from the table to be the i^{th} row and j^{th} column where i is the leftmost four bits of the byte, and j the rightmost four.

The purpose of this transformation is to introduce non-linearity to what is effectively a substitution cipher. Without this, the cipher would be susceptible to a differential analysis attack, meaning an attacker can exploit the difference between two plaintext and ciphertext pairs.

ShiftRows is a simple transformation the cyclically shifts the last three rows of the state with given offsets. Given a byte $b_{i,j}$ in a matrix, its new position after transformation is $b_{i,j} = b_{i,(c+i) \mod 4}, 0 \leq c < 4$. The first row is unaffected.

$$State = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

$$State' = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix}$$

MixColumns handles each column in a state as 4-byte words, and considers them as polynomials over the finite field GF(2^8) multiplied modulo $x^4 + 1$ with the polynomial $a(x) = \texttt{03}x^3 + \texttt{01}x^2 + \texttt{01}x + \texttt{02}$. $a(x)$ is represented as a matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Given a column c, we get:

$$\begin{bmatrix} b_{0,c} \\ b_{1,c} \\ b_{2,c} \\ b_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} b'_{0,c} \\ b'_{1,c} \\ b'_{2,c} \\ b'_{3,c} \end{bmatrix}$$

The purpose of `MixColumns` along with `ShiftRows` is to introduce a higher level of entropy in the message space where, due to the fundamentals of natural languages, low entropy distribution is highly likely.

AddRoundKey uses a bitwise XOR operation to add the current round key to the state. As with `MixColumns`, the state is handled column-by-column in 4-byte words, which are XOR'd with the matching 4-byte words in a 16-byte round key block.

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,1} & k_{1,2} & k_{1,3} & k_{1,0} \\ k_{2,2} & k_{2,3} & k_{2,0} & k_{2,1} \\ k_{3,3} & k_{3,0} & k_{3,1} & k_{3,2} \end{bmatrix}$$

Which, for a given column c, equates to $[b_{0,c}, b_{1,c}, b_{2,c}, b_{3,c}] \oplus [k_{0,c}, k_{1,c}, k_{2,c}, k_{3,c}]$.

The purpose of this transformation is apparent: apply the secret to the message.

4.2.3.2 Keys

Keys are input or generated as random n-bit byte arrays, where $n \in \{128, 192, 256\}$. This key is converted in to a *key schedule* as defined by a key expansion method, which consists of p key blocks (known as round keys) where p is equal to the number of rounds for the given key size. When on the p^{th} round of encryption, the p^{th} key schedule block is added to the current state block (§4.2.3.1).

Taking each column, as usual, in a block as a 4-byte word and starting with word $(i+4)$ where the first four words are the initial key, there are four steps to generating a word to become part of the key schedule:

1. **RotWord** – similar to `ShiftRows`, it cyclically rotates a word $[b_0, b_1, b_2, b_3]$ in to $[b_1, b_2, b_3, b_0]$.
2. **SubWord** – `SubWord` replaces each byte with a corresponding value from the S-Box table, using the same logic as `SubBytes`.
3. **XOR** – the word is XOR'd with the $(i^{th} - 4)$ word.
4. **Rcon** – the Rcon table is a *round constant word array*, the matching columns of which are XOR'd with the current word.

However, this only occurs for the first word in each 4 word block of the key schedule, and for 256-bit keys the `RotWord` step is omitted completely. The remaining 3 words simply XOR the $(i^{th} - 1)$ and $(i^{th} - 4)$ blocks.

An excellent animation that covers all aspects of the AES process can be found at [21].

4.2.4 Modes of Operation

Like most block ciphers, AES has a number of *modes of operation.* The simplest of which is what we've been describing so far: divide the plaintext in to n-bit blocks (where the message length $m_l > n$ and transform each block. This is know as the electronic-codebook (ECB). There are four most-prevalent modes: ECB, CBC (Cipher Block Chaining), CFB (Cipher Feedback), and OFB (Output Feedback).

ECB mode is as above:

Where C is a ciphertext block, P and plaintext and E the encryption function,

$$C_i = E_k(P_i),\ P_i = D_k(C_i)$$

Blocks are enciphered independently of all other blocks, and so any errors in a block do not *propagate* throughout the rest of the ciphertext, meaning the majority of the ciphertext will still be able to be decrypted.

However, this independence of blocks means that malicious blocks can be substituted into a ciphertext. Alongside this, we are open to frequency analysis and other ciphertext-only attacks. ECB is generally not recommended for use in a production environment.

CBC mode involves XOR'ing the first plaintext block with a random bit string known as the initialisation vector (IV) and then for each block the plaintext is XOR'ed with the previous block.

$$C_0 = IV \text{ and for } i\text{th block } C_i = E_k(P_i \oplus C_{i-1}),\ P_i = D_k(C_i) \oplus C_{i-1}$$

CBC is dependent on the correct ordering of the block chain, consequently rearranging the blocks or modifying any will affect the output of decryption. While this is beneficial in preventing attacks, it has an affect on error propagation. As a block c_i is dependent on c_{i-1}, if c_{i-1} contains any error, the decryption of block c_i will also be affected, which also opens the algorithm up to attacks by altering the bits of c_{i-1}. However blocks c_{i+2} are not affected by errors in c_i, providing a form of error recovery.

CFB mode is similar to CBC in that it is effectively the reverse of the CBC operation and also making use of an initialisation vector.

$$C_0 = IV,\ C_i = E_k(C_{i-1}) \oplus P_i,\ P_i = E_k(C_{i-1} \oplus C_i$$

OFB, interestingly, turns the block cipher effectively in to a stream cipher.

$$C_i = P_i \oplus O_i, P_i = C_i \oplus O_i,$$
$$\text{where } O_i = E_k(I_i) \text{ and } I_0 = IV,\ I_i = O_{i-1}.$$

OFB excels over other modes with regards to the avoidance of error propagation and can recover from bit errors in blocks. Conversely, the *loss* of block bits results in the keystream alignment being damaged meaning decryption is not possible.

We will be implementing ECB, as it is clearer to explain, with a discussion on how to implement other modes. And because the purpose of this is not to develop a perfectly secure algorithm. The underlying algorithm is not affected by any of these modes, as each block is still encrypted and decrypted using the transformations listed in §4.2.3. This means that implementing these modes as either an option or a permanent modification is relatively easy in terms of development time.

4.2.5 Implementation

As with the public-key algorithms, the AES algorithm will be realised using Java so that it can be referenced later by the Enigma application.

4.2.5.1 A Note on Block Representation

As is apparently obvious and shown in the algorithm description, blocks are 4 by 4 arrays of bytes. However, copying parts of blocks (represented as Java `byte[]`) and running calculations on them is awkward to visualise as the blocks will have to be made up of multi-dimensional arrays. As such, throughout the implementation we will use both a multi-dimensional array block representation and a one-dimensional array representation, the difference of which is made clear wherever each is used. §4.2.5.4 defines two methods that will convert between the two representations.

4.2.5.2 Key Generation

Key generation is simple as keys are just random bit strings of a length equal to the desired key size. We have created an `enum` called `KeySize` (the idea of which is partially from a project identified by the package *watne.seis720.project*) which takes the key size as input to a constructor, and provides helper methods such as `getNumberOfRounds()` to provide a persistent method of retrieving the current requirements for the key size without having to have hard-coded values within transformations and methods.

Keys are represented as an object `Key` with constructor `Key(KeySize k)` meaning it is provided with a `KeySize` enum object which defines the size of the key to be generated.

```
public Key(KeySize k) throws DataFormatException {
  byte[] key = new byte[k.getKeySizeBytes()];
  this.ksize = k;

  SecureRandom rng = new SecureRandom();
  rng.nextBytes(key);
}
```

This is perhaps misleading as random number generators tend to take a seed and a length and return a random number. However in this case we provide the generator with a byte array the size of the key needed and it fills it with random data.

As the `Key` object represents a key within a session, it also provides a method to return the expanded version of the key, however this will be covered in §4.2.5.5.

4.2.5.3 Constants

The S-Boxes used by the `SubBytes` transformation can be computed on-the-fly, however given that the values are always the same and independent of any plaintext or ciphertext input, it makes little sense to do so. As such, we use pre-computed S-Boxes that are stored in a class `AES_Constants` as a multi-dimensional array.

Alongside the the S-Boxes, `AES_Constants` also stores the matrices used for the `MixColumn` and `InvMixColumn` transformation as they are also fixed, rendering calculation of them on-access redundant.

4.2.5.4 Common Utilities

State Representation varies between methods. As we said previously, occasionally it is more useful to work on blocks as one-dimensional arrays rather than multi-dimensional, and as such we will need utilities that convert between the two types: `arrayTo4xArray()` and `array4xToArray()`. They are both reasonably simple – the latter loops through each of the 4 rows of bytes and appends them to a 16-byte array using `System.arraycopy()`:

```
for (int i=0;i<4;i++) {
  System.arraycopy(array[i],0,array1x,(i*4),4);
}
```

The former does the same, but in reverse, copying each section of 4 bytes from a 16-byte array in to the matching row in a multi-dimensional array:

```
for (int i=0;i<4;i++) {
  System.arraycopy(array,(i*4),array4x[i],0,4);
}
```

Padding is required for messages that are not divisible by the block size, 16. Adding zero bytes to “fill out” a byte array will work, however after decryption how will we know how many padding bytes to remove? [22], section 6.1.1, defines the padding string as consisting of $8-(||M|| \mod 8)$ bytes with the value $8-(||M|| \mod 8)$ for a message M. For example, for a message of length 6, we get:

$$M' = M \mid\mid 0202$$

As the padding bytes are each equal the number of padding bytes used, we know how many to remove. However, in our case we are using block lengths of 16 and so our padding strings will consist of $16-(||M|| \mod 16)$ bytes equalling $16-(||M|| \mod 16)$.

Multiplication Over the Finite Field could be completed arithmetically, however it is far simpler and more efficient to convert it in to using bitwise arithmetic. For example, the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ can be represented as the byte `0x11b`, and thus addition over the finite field can be calculated through XOR’ing the byte with `0x11b`.

```java
public static byte FFMul(byte a, byte b) {
  byte r = 0;

  while (a != 0) {
    if ((a & 1) != 0) r = (byte)(r ^ b);

    // Repeatedly multiply by (1)
    b = (byte)(b << 1);

    // If the result is of degree 8
    // add m(x)
    if ((byte)(b & 0x80) != 0)
        b = (byte)(b ^ 0x1b);

    a = (byte)((a & 0xff) >> 1);
  }

    return r;
}
```

4.2.5.5 Transformations

§4.2.3 explicitly defines the transformation algorithms used here. Refer to that section for detailed, non-programmatic descriptions.

SubBytes We have already decided that S-Box values are to be pre-computed and stored statically, so retrieving the S-Box value for each byte in a bock (the purpose of `SubBytes`) is trivial.

```
for (int i=0;i<block.length;i++) {
  for (int j=0;j<4;j++) {
    block[i][j] = AES_Transformations.getSBoxValue(block[i][j]);
  }
}
```

Here we are looping through each row and then column of a block and retrieving the S-Box value using a helper function:

```
public static byte getSBoxValue(byte o) {
  // Get 4 left-most and right-most bits
  int i = ((o & 0xf0) >> 4);
  int j = (o & 0x0f);

  return AES_Constants.SBOX[i][j];
}
```

ShiftRows cyclically shifts the byte in the last three rows of a state block. With starting row $r = 0$ in the set $0, 1, 2, 3$, each row is shift r times.

```
public static byte[][] shiftRows(byte[][] state) {
  byte[][] new_state = new byte[4][4];

  // Keep first row
  new_state[0] = state[0];

  for (int i=1;i<=3;i++) {
    // Copy r columns to the end of the new state
    System.arraycopy(state[i],i,new_state[i],0,4-i);
    // Copy the 4-r columns to the start of the new state
    System.arraycopy(state[i],0,new_state[i],4-i,i);
  }

  return new_state;
}
```

MixColumns iterates through blocks column by column and considers each column as a four variable ploynomial. As such we multiply each byte in a column with the predefined `MixColumn` matrix, and then XOR all the multiplication results together.

```
for (int col=0;col<4;col++) {
  column[col] = AES_Utils.FFMul(AES_Constants.MIXCOL[col][0], state[0][i]) ^
    AES_Utils.FFMul(AES_Constants.MIXCOL[col][1], state[1][i]) ^
    AES_Utils.FFMul(AES_Constants.MIXCOL[col][2], state[2][i]) ^
    AES_Utils.FFMul(AES_Constants.MIXCOL[col][3], state[3][i]);
}
```

AddRoundKey uses an XOR operation to add the round key to the current state block. For a given row r, we iterate through the bytes in the expanded key (§4.2.5.5) with indexes in the interval $[16r, (16r) + 16]$ and XOR them in order with each byte in the block.

```
byte[] exp_key = key.getExpandedKey();

// Initial round starts from index 0
// All further rounds start from 16 bits * round, i.e.
// 16 bytes ahead of the last round
int index = r*16;

for (int col=0;col<4;col++) {
  for (int row=0;row<4;row++) {
    state[row][col] = (byte)(block[row][col]^exp_key[index++]);
  }
}
```

KeyExpansion is the most complex of the transformations, and is actually part of the Key class. The algorithm itself is defined fully in §4.2.3 However, as shown in [23], this can be done in one encapsulated method. Following along the same lines, we use single bytes rather than 4-byte words as defined in [20]. Our implementation of key expansion is a simplified (and more readable) version of the method found in [23].

```
// Make following FIPS-197 pseudo-code easier and use their conventions
int Nk = this.getKeySize().getKeySizeWords();
int Nr = this.getKeySize().getNumberOfRounds();
int Nb = 4;

this.expanded_key = new byte[Nb*4*(Nr+1)];

byte[] cur_word = new byte[4];

int i;
for(int j=4*Nk; j < 4*Nb*(Nr+1); j+=4) {
   i = j/4;

   // Get the next 4 bytes (word) of the key
   for (int k=0;k<4;k++) cur_word[k] = this.expanded_key[j-4+k];

   if (i % Nk == 0) {
     // Loop through the current word
     for (int k=0;k<4;k++) {
       // Determine which byte of the word to use
        byte temp = cur_word[(k==3) ? 0 : k+1];
        // Determine rcon value
        byte rcon = (k == 0) ? AES_Constants.RCON[(i/Nk)-1] : 0;
        // xor each byte in the word with temp value and rcon value
        cur_word[k] = (byte)(AES_Transformations.getSBoxValue(temp) ^ rcon);
     }
   // As defined in FIPS-197...
   // For 256-bit keys we apply SubWord() to
   // expanded_key[i-1] before xoring below
   } else if ((Nk==8) && ((i%Nk)==4)) {
     for (int k=0;k<4;k++)
       cur_word[k] = AES_Transformations.getSBoxValue(cur_word[k]);
   }

   // The actual work..
   // xor each byte of the current word with the
```

```
    // matching byte in the 4 words behind
    for (int k=0;k<4;k++) {
      this.expanded_key[j+k] = (byte)(this.expanded_key[j - 4*Nk + k]
                                  ^ cur_word[k]);
    }
}
```

4.2.5.6 Algorithm

Now that the transformations and common utilities have been defined, it is trivial to implement the actual algorithm that executes the rounds on each block.

We define a method encrypt() that given a byte array will return an enciphered byte array by looping through each block of the given plaintext and enciphering it before placing it back in to a final ciphertext byte array at the relevant array index. The work is done by a method cipher():

```
private byte[] cipher(byte[] block) throws DataFormatException {
  // Conver the given 1x16 byte array in to a 4x4 array that
  // can be used by the transformation functions
  byte[][] state = AES_Utils.arrayTo4xArray(block);

  state = AES_Transformations.addRoundKey(state, this.getKey(), 0);

  // Iterate over the block for the number of rounds defined by the
  // type of key
  for (int r=1;r<this.getKey().getKeySize().getNumberOfRounds();r++) {
    state = AES_Transformations.subBytes(state);
    state = AES_Transformations.shiftRows(state);
    state = AES_Transformations.mixColumns(state);
    state = AES_Transformations.addRoundKey(state, this.getKey(), r);
  }

  // The final round excludes the mix columns transformation
  state = AES_Transformations.subBytes(state);
  state = AES_Transformations.shiftRows(state);
  state = AES_Transformations.addRoundKey(state, this.getKey(),
                                          this.getKey()
                                             .getKeySize()
                                             .getNumberOfRounds());

  return AES_Utils.array4xToArray(state);
}
```

decrypt() and invcipher() work identically, except invcipher() carries out the inverse transformations defined in §4.2.5.5.

These methods are contained within an object AES, which upon instantiation requires the setting of a plain- or cipher-text and a key.

4.2.6 Summary

As we can see, AES is relatively simple to implement in an efficient way. Now that we have an asymmetric algorithm and a symmetric algorithm, we need to implement a method of identifying entities.

Chapter 5

Identification and Authentication

5.1 Overview and Intentions

In this chapter we will be discussing the methods used by one entity to verify that the identity given by another is valid. This is often referred to as identification, or entity authentication. We say that the *verifier* is requesting assurance that the *claimant's* identity is actually what the claimant has defined. We also must define *timeliness* as part of entity authentication: the claimant's identity can only be verified during the execution of the identification protocol in real-time, this is known as a *lively correspondence.*

The main type of authentication that we will be looking at, particularly for the Enigma application, is *authenticated key establishment.* Often entity authentication protocols are used not only to verify identity, but also to further secure transmissions. For example, as we have discussed extensively, keys need to be shared securely and thus we also need methods for proving that a received key is from the entity with which we want to communicate.

5.2 Digital Signatures

5.2.1 Overview

A digital signature is often compared to real life signatures, a verification of identity, however this is not entirely true. A digital signature is comparable to a handwritten signature in the way that someone could handwrite a signature on a document to prove that it is valid and (perhaps in this example) written by them. The use of digital signatures extends further than this, and can be defined in terms of information security as listed in §2:

1. Authentication – the document comes from the expected entity (assuming the public keys can be trusted).
2. Integrity – a digital signature acts as a way of checking that the received data matches the data that was signed and transmitted.

3. Non-repudiation – a signed document cannot be revoked and the sending entity cannot claim they did not sign it.

This of course introduces a chicken-and-egg issue: how do you verify the identity of someone without having them digitally sign proof of their identity? This is solved by the use of trusted third parties and certificates (§5.3) – that is to say, having *another entity* sign proof of the sender's identity.

There are two overall types of digital signature algorithms: those with *appendices*, and those with *message recovery.* Generally signatures with appendix are more common.

1. With Appendix – reliant on cryptographic hash functions to provide message summaries that can be signed and then compared to summaries of the received document.
2. With Message Recovery – the message signed can be recovered from the signature, which is typically of use only for short messages.

We will be using a scheme with appendix: RSA.

An example of a transmission between two entities, Alice and Bob, is given to show the process of signing and verifying messages.

1. Alice creates a message M and signs it using her private key known only to her, producing signature S, sending both to Bob.
2. Bob receives $\{S, M\}$ and retrieves Alice's public key.
3. Bob verifies that M is unmodified (integrity) and from Alice (authentication) by comparing it to S using a pre-defined method.

Signatures are a simple but secure method of ensuring information security.

5.2.2 RSA

The RSA digital signature scheme was the first feasible scheme to be introduced, and remains the most popular. It works in very much the same way as RSA public-key encryption/decryption: through the generation of an exponent and modulus to be used in a bijective formula where the exponents are reversed for encryption/decryption.

Why use RSA? It is the most prevalent algorithm for message signing, and as we have covered the theory already it should be easy to understand and link in with our existing key infrastructure.

5.2.2.1 Algorithm

Key generation is the same as in §3.6.1.3.

1. Signing (Alice):
   (a) Convert message m in to an integer representative.
   (b) Calculate $s = m^d \mod N$, where d is the private exponent, and N the modulus.
   (c) Attach s to the message m and transmit $\{s, m\}$.
2. Verification (Bob):
   (a) Receive $\{s, m\}$ and obtain Alice's public key.
   (b) Calculate $em = s^e \mod N$, where e is Alice's public exponent.
   (c) If em matches m, then m is valid.

5.2.2.2 Implementation

This is trivial to implement, however as with RSA encryption/decryption the standard algorithm is susceptible to several attacks. Our previous scheme – OAEP – is not applicable for use in digital signatures and so we will be using a scheme known as RSA-PSS: RSA Probabilistic Signature Scheme. Approved in 2003 as an RFC[16], it is now the accepted standard to be used for digital signatures utilising the RSA algorithm.

The scheme adds two well-defined methods in to the algorithm: `EMSA-PSS-ENCODE`, an algorithm to format data before mathematically signing it, and `EMSA-PSS-VERIFY` an algorithm that verifies the signature *after* it has been mathematically verified using RSA. To better describe the scheme, we also define two methods as displayed in the RFC[16]: `RSASP1` produces a signature representative of a message m, and `RSAVP1` returns a message representation of a given signature s.

Signing a message m now works as so:

1. Encode the message: `EMSA-PSS-ENCODE(m)`.
2. Convert the message to an representative integer form: `OS2IP(m)`.
3. Using a private key p, calculate the signature representative s using RSA: `RSASP1(p,m)`.
4. Convert the signature (integer) representative back to a byte string: `I2OSP(s)`.

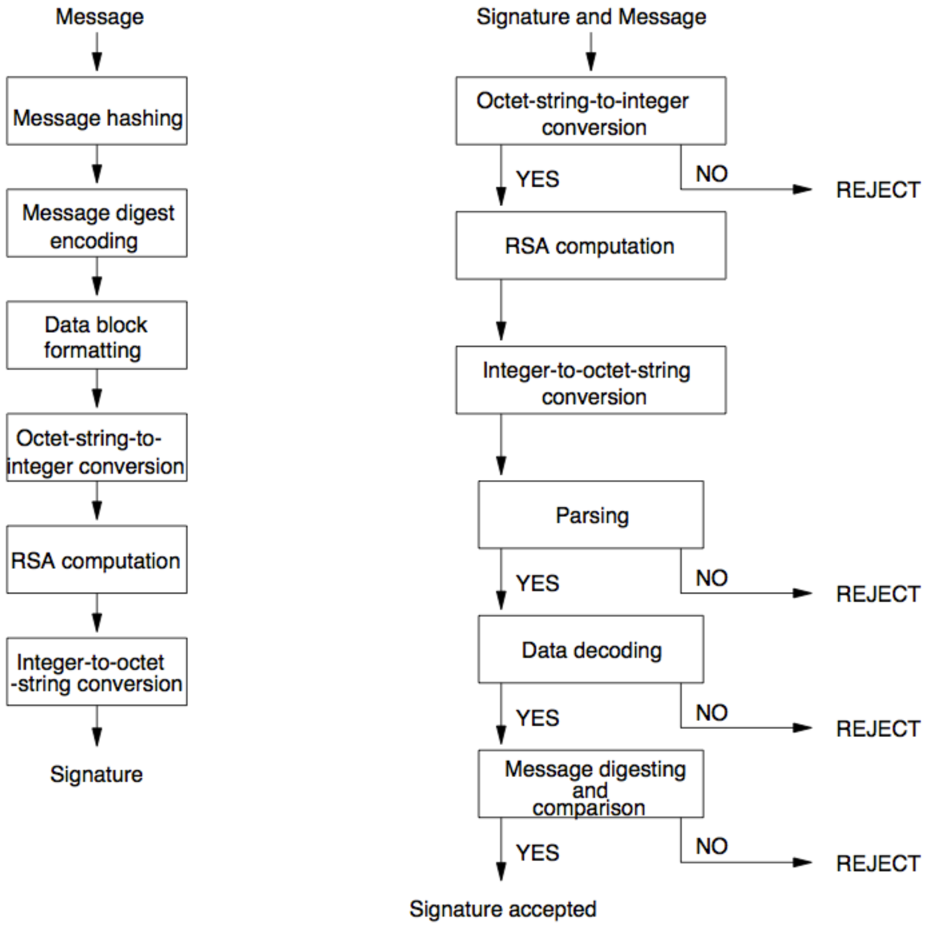
Verification is the reverse, with the method change as described above. Given a signature s:

1. Convert s into an integer representative: `OS2IP(s)`.
2. Using public key p, calculate the message representative m: `RSAVP1(p,s)`

3. Convert m to a byte string: `I2OSP(m)`.

4. Verify the byte string using `EMSA-PSS-VERIFY`.

Indeed, [3] shares a simplified diagram of how the processes work:



[16] makes one (unstated) point: good reference documentation for an algorithm will almost always result in a robust implementation. This RFC gives an excellent overview and exact requirements for the algorithm, meaning that our implementation process is relatively pain-free.

RSA Primitives are of course the simplest. Recall that given an exponent e – be it private or public – and modulus N, we have $m = c^e \mod N$. Interestingly, this means we only have to implement the actual code once and the other primitive can simply override this method for naming convention as both public and private exponents are large integers and can be swapped interchangeably.

```
private BigInteger RSASP1(BigInteger m) {
  // 2. Let s = m^d mod n.
  BigInteger s = m.modPow(E, N);
  return s;
```

```java
}

private BigInteger RSAVP1(BigInteger s) {
  // 2. Let m = s^e mod n.
  return this.RSASP1(s);
}
```

Once again we are using the `BigInteger` class for storing large integers (see §3.4.1.3).

Before discussing the implementation of the EMSA–PSS functionality, we must first define how they actually work.

EMSA–PSS–ENCODE differs from the original signature-with-appendix scheme created by Bellare and Rogaway, almost entirely in the small details. For example, applying a hash function instead of a mask generation function.

The structure of the final encoded string is somewhat similar to that in RSA–OAEP [16].

```
                         +-----------+
                         |     M     |
                         +-----------+
                               |
                               V
                             Hash
                               |
                               V
                 +--------+----------+----------+
            M' = |Padding1|  mHash   |   salt   |
                 +--------+----------+----------+
                                |
      +--------+----------+     V
DB =  |Padding2|   salt   |    Hash
      +--------+----------+     |
                |               |
                V               |    +--+
               xor <--- MGF <---|    |bc|
                |               |    +--+
                |               |      |
                V               V      V
      +-------------------+----------+--+
EM =  |    maskedDB       |     H    |bc|
      +-------------------+----------+--+
```

1. A message M is hashed, and then this hash is concatenated with eight zero-bytes (`Padding1`) and a random-byte string, with length equal to the digest length of the hash function, known as a salt.

2. A byte array DB is created, consisting of a padding string and the salt, separated by the byte `0x01`.

3. A mask generation function (§3.6.1.7) is used to mask the hashed message to a length of (the maximum size `BigInteger` - the maximum hash length - 1), which is then XOR'd with DB.

4. The final string is the masked DB with the hashed message and a byte `0xbc`.

Due to the length of the code, it is recommended that it is viewed electronically, and can be found at *com.cyanoryx.uni.crypto.rsa.RSA_PSS*.

Given the message M and encoded message EM, EMSA–PSS-VERIFY works similarly in that it encodes the given message and compares the computed signatures to the one received, along with some interstitial error checking to remove unnecessary calculation if the verification fails early on (for example, in correct padding separator bytes).

1. Hash the message so $H = M$, and check to see if the rightmost byte of EM is `0xbc`.
2. Create *maskedDB* to be (the maximum size `BigInteger` - the maximum hash length - 1) bits of EM and the next (maximum hash length) bits of M.
3. Mask the hash of M to be (the maximum size `BigInteger` - the maximum hash length - 1) bits, and XOR it with *maskedDB*, creating DB.
4. Let `salt` be the last n bytes of of DB where n is a predefined salt length.
5. Create the byte array M' such that
   $M =$ `(0x) 00 00 00 00 00 00 00 00 || Hash(`M`) || salt`.
6. Let $H' = Hash(M')$ and if $H = H'$, the signature is valid, otherwise not.

Keys are handled using the same infrastructure as RSA–OAEP. A `Key` object with descendants `PrivateKey` and `PublicKey` store the exponent and modulus, and is passed in when the `RSA_PSS` object is first instantiated. This means that keys stored on disk or in memory can be imported without the developer parsing them manually.

5.3 Certificates

5.3.1 Overview

Now that we understand digital signatures, we can address an issue with their use: how can you verify that the public key you are using is the key belonging to the entity you believe you are receiving communications from. The scenario is as so:

1. Alice creates and signs a message m, sending Bob the signature s with it: $\{s, m\}$.
2. Mallory intercepts the message, modifies it, and signs it using her own private key and sends it to Bob.
3. Bob received the message, and obtains Alice's public key. However, Mallory – the means through which she does it are not relevant – has replaced Alice's public key with her own.

4. Bob now verifies Mallory's message, believing that it has come from Alice and has not been changed.

A commonly used concept in cryptography is the *trusted third party* (TTP). In this case, an entity with whom we place trust to act as an arbiter or facilitator between us and other entities to mutually verify identity. In the case of certificates, a TTP is known as a Certificate Authority (CA) and our trust in them is based on a *chain* that is bundled with operating systems and web browsers. This chain consists of the public keys of the CAs, which can be used to verify the keys of other entities, however we will discuss this further below. These trusted third parties are the most widely used solution to the problem of ensuring that received public-keys match with the entity we expect.

Keypair owners pay Certificate Authorities to accept their public keys and proofs of identity (some form of ID, bank statements, proof of address, etc.) so that they can sign the keypair owner's public key and distribute the signature so that other entities may verify this signature and key, thus proving it belongs to the designated person.

However, we need a way of distributing all this information in a standardised way – certificates. The current standard used for structuring the data stored in certificates is known as *X.509v3*[24]. This defines a rigid structure that X.509 certificates must follow, such as in what order to include the subject's (entity's) name, key, signature, etc. and how to do so.

5.3.2 Enigma Certificates

As part of our testbed we will be implementing a version of the X.509v1 specification[25]:

```
Cert { Algorithm, Signature },
CertDetails {
  Issuer,
  Serial,
  Validity { notBefore, notAfter },
  UniqIdent,
  SubjectInfo { Name, Algorithm, Public Key }
}
```

We create a class `Certificate` for storing this information, as well as handling certificate files and exporting them. Within this class, Certificates are stored locally within private `byte[]` fields, and are passed to the class using a `java.util.HashTable<String,byte[]>` so that multiple parameter handling is not needed on either end.

Outside of the class, certificates are stored in a pre-defined ASCII format (items enclosed in square brackets represent a field to be filled with data):

```
-----BEGIN-ENIGMA-CERTIFICATE----- \n
[SignatureAlgorithm],[Signature]//[Issue]|[Serial]|
[ValidityNotBefore],[ValidityNotAfter]|[UniqIdent]|
[SubjectName],[SubjectAlgorithm],[SubjectKey] \n
-----END-ENIGMA-CERTIFICATE-----
```

We provide two constructors: `Certificate(String c)` that automatically parses the data provided in to a `Certificate` object, and `Certificate(File c)` that reads a file in then passes it to `Certificate(String c)` for processing.

Alongside this `toString()` is overridden to output the certificate in a writeable form, and `toReadable()` to output the certificate in a text form that is readable by a user.

The details of this class can be found at *com.cyanoryx.uni.crypto.cert.Certificate*, and are not listed here to the simplicity of its operation.

5.3.2.1 Certificate Authority

This is no use, however, without a trusted third party to sign the keys in the certificate. We create a utility class, `CertificateAuthority`, that signs keys and verifies certificates using the `Certificate` class and RSA-PSS.

```
public boolean verify(Certificate cert) throws DataFormatException {
  byte[] signature = cert.getSignature();
  byte[] key       = cert.getSubject_key();

  RSA_PSS rsa = new RSA_PSS(this.pub);
  return rsa.verify(key, signature);
}

public byte[] sign(byte[] key) throws DataFormatException {
  if (this.priv==null) throw new DataFormatException("No private key found");

  RSA_PSS rsa = new RSA_PSS(this.priv);
  return rsa.sign(key);
}
```

5.4 Summary

With certificates and digital signatures, the flow of encryption and decryption when sharing messages is different:

1. Bob sends his public-key certificate to Alice.
2. Alice verifies Bob's public key in his certificate by using the certificate authority's public key.
3. Alice generates a session key k, and encrypts it using Bob's public key and transmits it to him.
4. Bob decrypts the session key and now both entities have a share secret key.

Chapter 6

Enigma: A Testbed

6.1 Overview and Intentions

The intention of this project is to produce an application that allows two users to securely identify and authenticate one another, and communicate with text messages via a presumed-insecure network. However, the primary *goal* is to create a testbed that allows any cryptographic algorithms to be integrated in place, to provide a platform for further analysis, such as comparison of run-times between open and closed source implementations.

6.2 Engineering Methodologies and Planning

6.2.1 Methodology

The main focus of the development process was to break the application down in to its component pieces, and iteratively develop them so that minimalist functionality sets could be produced and then combined in the shortest possible times. The overarching category for this style of production is known as *agile development*: a practice based around iterative and incremental development. More specifically, a subset of agile development will be used – Scrum. The goals of Scrum fit neatly with the desired development cycle in that programming is done in ”sprints,” with a deadline organised at the start of these sprints and a set of tasks that must be completed by the end. A sprint can last anywhere between one week and one month. This, combined with test driven development (TDD, writing tests for functionality before producing the functionality), produces a set of components matching a well-defined requirements document that can be combined in to the final product.

However, without results, methodologies are meaningless. One should not concentrate too heavily on explicitly following a development process, particularly when working alone, otherwise work output can be reduced significantly by the overhead and “red tape.” Because of this, the methodology of development will not be encountered again in this document, however it is of interest to know the basics as above.

6.2.2 Documentation

Documentation is an extremely important part of software engineering. It is a broad concept, and encompasses: requirements and specification, design overview, technical details, user manuals and even marketing information. However, in terms of the software development process, it is only the first three that are of direct importance.

It is said a program listing should be documentation unto itself: the programming style should allow an overall structure and purpose to be easily determined from examining the code [26]. However, this is an idealistic view and design and technical documentation are of great importance, not only for the developer currently producing the software, but for those possible developers in the future who have to maintain the code.

As a major component of this project is to develop a library of cryptographic algorithms, having a clear and accurate listing of all available methods in the API is vital. Conveniently, Java and the Eclipse IDE are tightly integrated with *JavaDoc*[1], a documentation generator that automatically produces standardised API documentation in HTML format based on comments inserted in the code. The comment blocks – distinguished using the format `/** ... */` – contain a method description, and a number of control sequences that give detailed information about what the method returns, the parameters it takes and other details such as exceptions thrown.

JavaDoc outputs are bundled with the appropriate packages in the file tree, and a manual is found in the appendices. A requirements specification has been compiled in §*A*.

6.3 Application Development

6.3.1 User Interface

*Packages covered in this section: com.cyanoryx.uni.enigma.gui.**

6.3.1.1 GUI Frameworks

SWT – the Standard Widget Toolkit – initially created by IBM and now maintained by the Eclipse Foundation, it is a modern alternative to Swing. Initially SWT was used (predominantly in tests) for the Enigma application, however as development continued it became more apparent that it was introducing a level of complexity to the interface code so much so that development slowed to an unmanageable speed. SWT benefits from its use of native component libraries, generally resulting in a more congruent user experience, however it falls down when it comes to ease of development.

Pros and cons of Swing

[1] http://java.sun.com/j2se/javadoc

Pros	Cons
Part of the JDK, meaning there's no need for native system libraries	The look and feel does not always match well with the native system
No differences in development between platforms and systems	
Very good documentation available, particularly from Sun	
Mature and well supported	
Is supported by official Java extensions like OpenGL	

Pros and cons of SWT

Pros	Cons
Uses native components	Native libraries must be available for all supported systems
Also supported by the Eclipse editor	Some native resources are not available on other systems, and so portability can be damaged
Documentation is good (though not as good as Swing)	Requires manual management of resources rather than, for example, SWT disposing windows, they must be done by the developer.
SWT programs can also integrate Swing components	SWT requires separate libraries to be distributed for 32- and 64-bit systems.
SWT is supposedly faster at rendering than Swing, though only minimally	

SWT appears initially simpler to use thanks to its use of the Model–View–Controller design pattern, the ability to "plug in" different look and feel settings, and so on, however it introduces the necessity to manage resources manually, rather than following the Java standard method of automatic resource disposal (known as the garbage collector).

Swing was selected as the framework to be used, primarily due to its current prevalence over SWT. Purely comparing speed statistics, SWT comes out on top, however the community behind Swing is far larger and more important than the downsides introduced by using Swing, and clearly the balance of pros and cons is in Swing's favour.

6.3.1.2 Designs

Being an application purely for testing purposes, the interface can have a certain leniency in terms of usability as those utilising it will likely have a good understanding of systems anyway. However, more importantly, the interface should be "invisible," meaning that the user should not have to think about using it and can focus on the task they are trying to complete – e.g. comparing encryption algorithms.

The application will have four main windows:

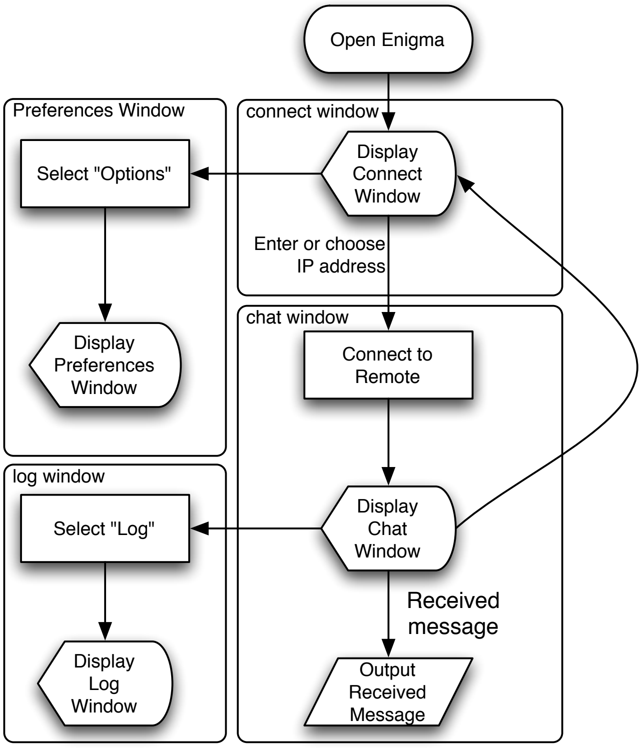


Figure 6.1: Overall Application Flow

1. Connect Window – The main starting point of the application. Users will enter the IP address of another Enigma server, or select an address from a "recently connected" menu.
2. Chat Window – Where the actual communications will take place. It will consist of a main listing box to display a message history for the current conversation, and have an input box for sending messages. It will also contain options for: Regenerating session keys, changing the cipher type, changing the agreement cipher type, and viewing the log window for this session.
3. Log Window – Can be used to log info, warning and error messages that do not cause a general program crash.
4. Preferences – Used to modify basic preferences, like default ciphers, username, etc.

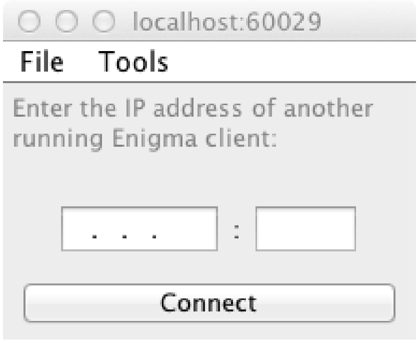
Figure 6.1 shows a basic overview of how the application will be used.

As developed with Swing, the four main windows look as so (on Mac OS X):

6.3.1.3 Connect Window

The Connect window comprises of two main pieces of logic:

1. Server creation.



Enter the IP address of another running Enigma client:

2. Connection making.

Firstly, the Connect window is the main focal point of the application and as such the creation of a `Connect` object signifies the start of the Enigma application. `Connect#Connect()` creates a new `Server` object with a random port number and initiates the process of creating the UI for the window.

As with most Swing-utilising classes, event handlers are used to process events triggered by user input – such as a button press – and in this case the main event handler will be that of the connect button. When a valid IP address is input and the button is clicked, the `connect()` method is run:

```
private void connect(String address, String remote_port) {
  try {
    // Display a loading screen
    Connect.this.showLoading();

    // As we are the initialiser,
    // generate a session ID
    String id = ""+(new Random().nextInt(100));

    // Create a new Session to store data for the remote server
    Session session = Server.createClient(address,
                      remote_port,
                      ""+port,
                      new User("remote user"),
                      id);

    // Store the Session in the server's session index
    Connect.this.server.getSessionIndex().addSession(session);

    // Send our desired agreement method
    session.sendAuth("method",
                     "agreement",
                     new AppPrefs().getPrefs()
                                   .get("default_asym_cipher","RSA"),
                                        id);

    // Send our public key and certificate
    session.sendAuth("cert",
                     "agreement",
                     Base64.encodeBytes(new Certificate(
                                            new File("./cert"))
                                            .toString()
                                            .getBytes()),
                     id);

    // Store the remote server's IP address
    new AppPrefs().getPrefs().put("last_connections",
                   address+":"+port+";"+new AppPrefs().getPrefs()
                                   .get("last_connections",""));
  } catch (Exception e) {
    [...] // Display an error
  } finally {
    try {
      // Always reinstate the UI
      Connect.this.recreateUI();
    } catch (ParseException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

Here we make use of a method `Server#createClient()`, which is a bit of a misnomer based around a legacy concept within the application where a local user would run both a server and a “client,” for sending and receiving data, respectively. This was designed as such due to some restrictions on the use of sockets, however this was removed for simplicity and due to the confusion introduced by running a “client” locally.

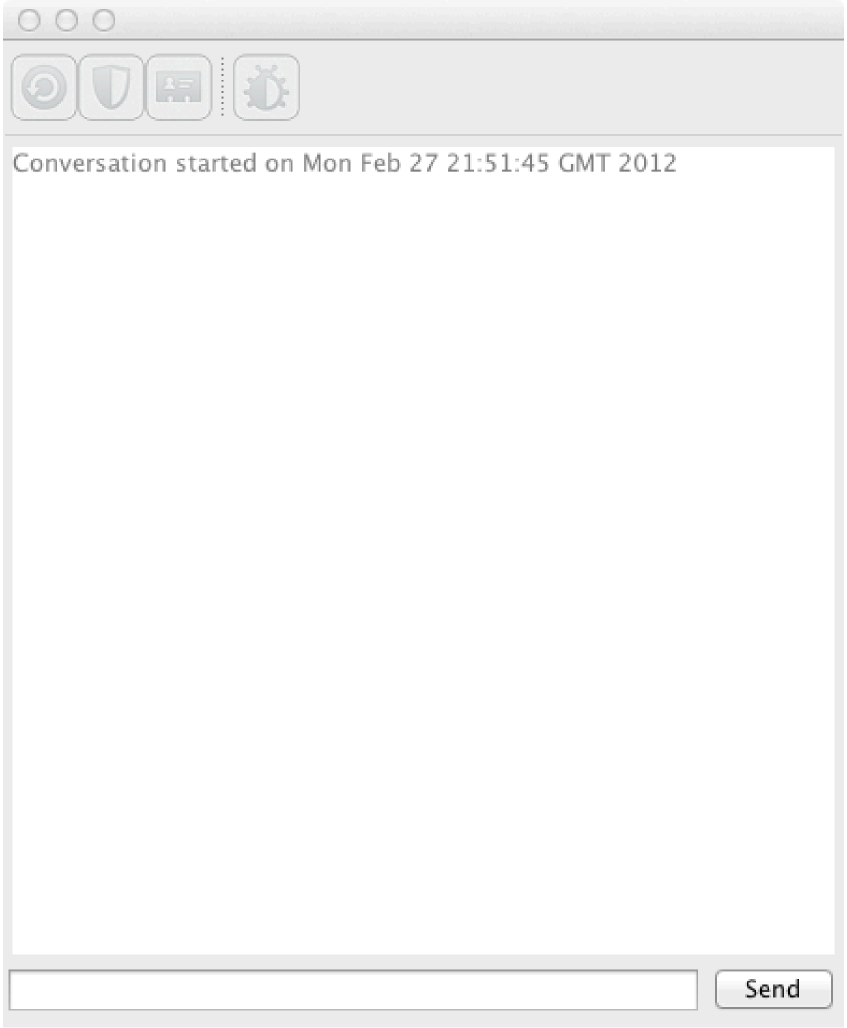
`Server#createClient()` is a simple, static method that creates a `Session` object with the supplied attributes and returns it, along with a `Conversation` object for the chat window.

6.3.1.4 Chat Window

The conversation has three tasks:

1. Display received messages.
2. Take the user’s input and send the message to the relevant `Session` object for encryption and transmission, and then display that message in the window (if sent).
3. Provide access to the following functionality:
   (a) Opening/closing the log window.
   (b) Regenerating the session keys.
   (c) Viewing the remote user’s certificate.
   (d) Toggling encryption on and off.

Each are relatively simple as most of the hard work is handed off to other objects within the `Server`’s jurisdiction.



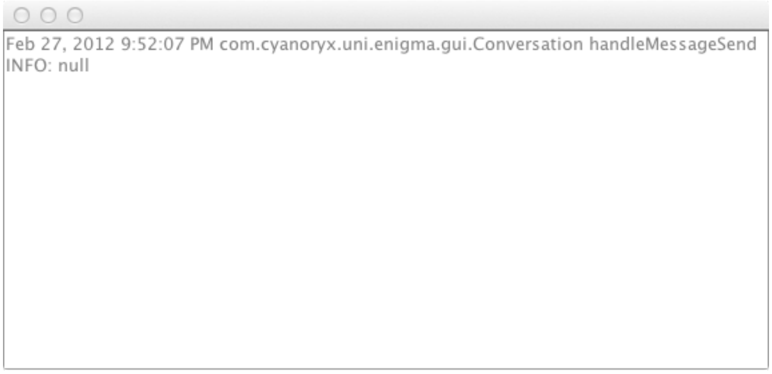
Conversation started on Mon Feb 27 21:51:45 GMT 2012

FIGURE 6.2: Chat Window

The display of messages is implemented using a `JTextPane` which, unlike a `JTextArea`, honours the use of font styling, colours and post-creation programmatic string insertion. The `updateMessage()` method takes a user name and a message:

```
public synchronized void updateMessage(String name, String message)
                 throws BadLocationException {
  StyledDocument d = messages.getStyledDocument();

      SimpleAttributeSet kw = new SimpleAttributeSet();
      StyleConstants.setBold(kw,true);

  d.insertString(d.getLength(),name+": ",kw);
  // Use a blank attribute set for the actual message text
  d.insertString(d.getLength(), message+"\n", new SimpleAttributeSet());
}
```

Two new Java classes are required: `StyledDocument` and `SimpleAttributeSet`. The former converts the messages `JTextPane` in to an object that can be formatted, and the latter is a set of attributes that can be applied as formatting to the text we are appending to the text pane.



```
Feb 27, 2012 9:52:07 PM com.cyanoryx.uni.enigma.gui.Conversation handleMessageSend
INFO: null
```

FIGURE 6.3: Log Viewer

`Conversation` windows are somewhat different from other interfaces in that they are persistent to a connection – conversations windows are stored in `Session` objects (as we will see later). Because of this, when a message is received, the handler retrieves the `Conversation` object and calls `updateMessage()`.

Sending messages is handled entirely by `Session` objects, and so when a user enters text and hits enter or clicks "send," the `Session#sendMessage()` method is called.

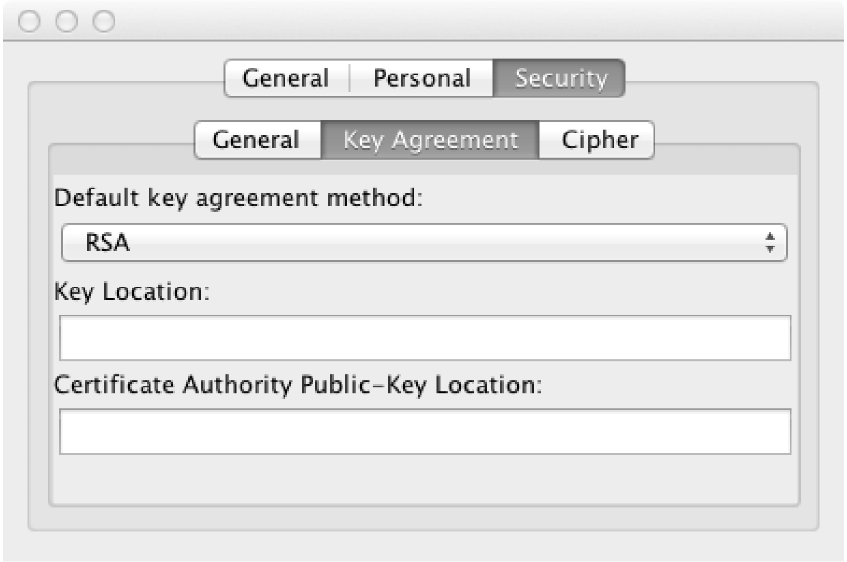
Displaying the log window, viewing the remote user's certificate and regenerating the session keys are all similarly simplistic in their implementation. The latter utilises the `Session#sendAuth()` method to regenerate a key and send it, however this is discussed in detail later. The other two buttons simply toggle the current display status of the two windows.

Toggling the encryption status is slightly more complex, however. The current status of encryption for the conversation must be checked, and turning it on or off depending on the result, also checking if the user allows for unencrypted conversations.

6.3.1.5 Log Window

The log window itself is a simple setup (`LogWindow`): a Swing `JFrame` wrapper with a disabled `JTextArea` and a method for handling input to be appended to the text area. The complexity behind it, however, is in the `LogHandler` class. `LogHandler` extends `java.util.logging.Handler`, but why not just have a `LogWindow` object and have each class manually update the text area? Using a `Handler` implementation is all about *extensibility*: it provides built-in capability for handling logging level settings (e.g. only log warnings, information, errors, etc.), along with applying formatters and filters. In the current version of Enigma, only the `Conversation` class needs to log information and so the log handler does little data modification and simply passes the messages on to a `LogWindow` object.

For the full code behind the logging system, see *com.cyanoryx.uni.enigma.gui.Log*.java.*



Default key agreement method:
RSA

Key Location:

Certificate Authority Public-Key Location:

FIGURE 6.4: Preferences

6.3.1.6 Preferences Window

User preference changed are handled by a single, unified preferences window, created by a `com.cyanoryx.uni.enigma.gui.Preferences` class (which should be distinguished from `Preferences` in the *java.util.prefs* namespace). It is another simple class that implements a `JFrame` containing multiple `JTabbedPane` elements separating out preferences in to their relevant categories.

Upon loading, each preference component is created with its value equal to the currently stored value, or if there is none, a default value. Each preference is governed by an event listener appropriate to the data type of the option:

1. `ItemListener` – boolean preferences use `ItemListener#itemStateChanged()` to register for a checkbox state change.
2. `ActionListener` – preferences with multiple choices implemented as drop down combo boxes register with `ActionListener#actionPerformed()` to monitor item selection.
3. `DocumentListener` – string based preferences, such as user name, are monitored using the update registering methods in `DocumentListener` to detect updated, deleted or inserted text.

When an event is generated, the updated preference is stored, meaning that the user never has to press any “save” or “apply” buttons.

In the future, it would perhaps be interesting to retrieve the list of available settings and their datatypes, and automatically generate a preference interface based on this. However, this may pose problems for preferences with multiple choices (like in drop down boxes), as all the options will not be saved in the preferences storage.

See §6.3.2 for the actual implementation of preference storage.

Each window was developed using the Swing framework, as mentioned, but the ”look and feel” of the components were designated to match the JDK’s interpretation of the system’s theme, meaning that the components will be very similar to those used in native applications thus making the application more portable, at least in terms of appearance.

6.3.2 Preferences

It is necessary to have some persistent data between application sessions, for example the user’s display name and their cipher preferences. There are many simple and many complex methodologies to solve this problem, the simplest seemingly being saving preferences to a text file with a custom format. However, as with any proprietary format it will be non-portable from the outset, and likely difficult to maintain in the future. Large amounts of “boiler–plate” code will be required to manage aspects like text-encoding, storage location, file formats, and so on which will unnecessarily add to development time and likely create bugs.

The JDK comes with a class *com.util.prefs.Preferences* for storing a collection of preference data. `Preferences` stores data in a key-value persistent backing store that is independent of the system implementation. Given a “node” name, `Preferences` will load data from the appropriate storage implementation, depending on the operating system. For example, it may store preferences in flat files, registries or even databases. This has one primary benefit: the developer does not need to concern themselves with the details of storage, but merely implement an application that creates a `Preferences` object given a node name and `Preferences` will handle the details.

More specifically, `Preferences` has a static method *userNodeForPackage(String name)* which returns a `Preferences` object containing handles that reference the persistent backing store. This `Preferences` objects provides helper methods such as *get(String key, String default_val)* that will programmatically return the value for the key `key`, handling any errors and using the default value if necessary.

However, this introduce an integrity and consistency issue: what if the node name changes? Do we store the node name in a centralised location? (introducing something of a chicken and egg problem). Or perhaps assume that it will never need to change? In our case, a wrapper class was created that returned a `Preferences` object with a node name equal to `getClass()`. This way the node will always be named after the package identifier for the given class, which can be refactored if necessary.

```java
package com.cyanoryx.uni.enigma.utils;

import java.util.prefs.Preferences;

/**
 * Creates a preferences object using this class name as the node.
 * Allows us to refer to the same preference across multiple objects.
 *
 * @author adammulligan
 *
 */
public class AppPrefs {
  /**
   * Returns a Preferences object for this application
   *
   * @return
   */
  public Preferences getPrefs() {
    return Preferences.userNodeForPackage(getClass());
  }

  /**
   * Returns an array of the last 10 connections as IP addresses
   *
   * @return
   */
  public String[] getLastConnections() {
    String   ips         = this.getPrefs().get("last_connections", "");
    String[] connections = new String[10];

    String separator = ";";

    connections = ips.split(separator);

    return connections;
  }

  /**
   * Removes all previous connections saved
   *
   */
  public void clearLastConnections() {
   this.getPrefs().put("last_connections","");
  }
}
```

This code can be found in com.cyanoryx.uni.enigma.utils.AppPrefs

We have also written a helper method that returns the last 10 IP connections, which are stored upon connection as a concatenated `String` array.

6.3.2.1 Usage

```
Preferences p = new AppPrefs().getPrefs();
p.get("preference_name","test");
```

6.3.3 Internationalisation

Internationalisation, colloquially shortened to *i18n* as 18 is the number of characters between the first and last letters, of an application is adding the possibility of easily creating new language sets that allow components and text in the application to be translated and shown in another language while retaining the ability to switch back to other languages. The purpose is to remove the need for software changes, and introduce a preference or setting that points the application to appropriate language set.

In this case, the language sets are stored in `.properties` files:

```
conv.toolbar.regen=Regenerate Session Key
conv.toolbar.toggle_dh=Toggle Encryption
conv.toolbar.view_cert=View Certificate
conv.toolbar.view_log=View Log
```

As with the preferences, a helper class is necessary to access the locale information and abstract some of the more verbose syntax needed for getting language information. In this case, a class `Strings` has been created that on initialisation gets the locale preference and attempts to load the corresponding language set (known in `Locale` parlance as a bundle), and provides a translate method that takes a string identifier, and returns the matching language string from the current locale bundle.

```
package com.cyanoryx.uni.enigma.utils;

import java.text.MessageFormat;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.prefs.Preferences;


public class Strings {

  public static Locale[] SUPPORTED_LOCALES = {Locale.ENGLISH};

  private static ResourceBundle resourceBundle;
  private static MessageFormat formatter;

  /**
   * Return the localised string for the given identifier.
   *
   * @param messageName
   * @return
   */
  public static String translate(String messageName) {
    return resourceBundle.getString(messageName);
  }

  /**
   * Return the current locale
   *
   * @return
   */
  public static Locale getCurrentLocale() {
    return resourceBundle.getLocale();
  }

  /**
   * Initialise the object and load the language bundle
   * from preferences.
   *
   */
  public static void initialise() {
    Preferences prefs = new AppPrefs().getPrefs();
    Locale locale = new Locale(prefs.get("locale", "en"));
    loadBundle(locale);
  }

  public static void loadBundle(Locale locale) {
    resourceBundle = ResourceBundle.getBundle("Enigma", locale);
    formatter = new MessageFormat("");
    formatter.setLocale(locale);
  }
}
```

This code can be found in com.cyanoryx.uni.enigma.utils.Strings

6.3.4 Drawable Graphics

Some elements of the user interface require the inclusion of local images and icons. For example, in the chat window the menu bar buttons contain representative icons rather than text. However, as with strings of text, hard-coded references to resources generally results in excessive maintenance if the resource location changes. As a solution to this, we will create a lightweight class modelled around the *Android SDK* concept of "drawables." Drawables are graphic resources stored in the system, represented by an identifier which can be retrieved using the `Drawable` class.

```
package com.cyanoryx.uni.enigma.utils;

import javax.swing.ImageIcon;

/**
 * Basic implementation of asset management, similar to drawables used
 * in Android SDK development.
 *
 * @author adammulligan
 *
 */
public class Drawable {
  public static final String DRAWABLE_DIR="res/drawable/";

  /**
   * Returns an ImageIcon for the given resource, stored in
   * DRAWABLE_DIR
   *
   * @param name
   * @return
   */
  public static ImageIcon loadImage(String name) {
    return new ImageIcon(DRAWABLE_DIR + name);
  }
}
```

This code can be found in com.cyanoryx.uni.enigma.utils.Drawable

As an example: if we decide that a database is more suitable for storing graphics than local files, the `loadImage()` method can be modified to search the database for the identifiers, and all the (possibly) hundreds of references to images within other parts of the codebase will not have to change.

6.4 Protocol Implementation

*Packages covered in this section: com.cyanoryx.uni.enigma.net.**

6.4.1 Overview

The purpose of the Enigma application is to provide an example application to be used for testing cryptographic algorithms, however the *function* of the application is to provide a means of communication between two entities who are capable of running the same software. To do this we will be implementing a custom variation of the Jabber/XMPP protocols within a server that will be able to send and receive XML data streams.

Servers are simple to implement, particularly with Java's excellent networking support – simply open a port locally that accepts connections, and parse any input and return outputs where necessary. However, there are many other aspects of running a server that are not covered by this in any form: for example, handling multiple connections efficiently and reliably, maintaining message integrity, and so forth. Because of this, we will be creating a server that is extensible and advanced in how it deals with connections and input. Alongside this, by introducing XML parsing capabilities and a well-defined protocol we are able to improve reliability (by being able to detect malformed XML) and make handling different types of input easier (by have distinct sets of XML tags for specific purposes).

6.4.1.1 Goals and Objectives

1. Simplicity – the server should have components which are only strictly necessary. Very little time should be spent on input processing.
2. Easy to integrate with cryptographic algorithms – this is harder to quantify, however it should result in software where there are only a small handful of points where algorithms will need to be connected, reducing the amount of repetition.
3. Extensible – it should be easy to add new handlers in the event the protocol is modified.

For a full requirements specification, see §A.

6.4.2 Server Design

6.4.2.1 Server

The server is be split in to four major components:

1. Sessions – connections between two entities will be managed by `Session` objects stored in an index.
2. Packets – packets represent sections of an XML document are stored in a queue upon arrival, and are handled in a first come first served basis by the XML parser.

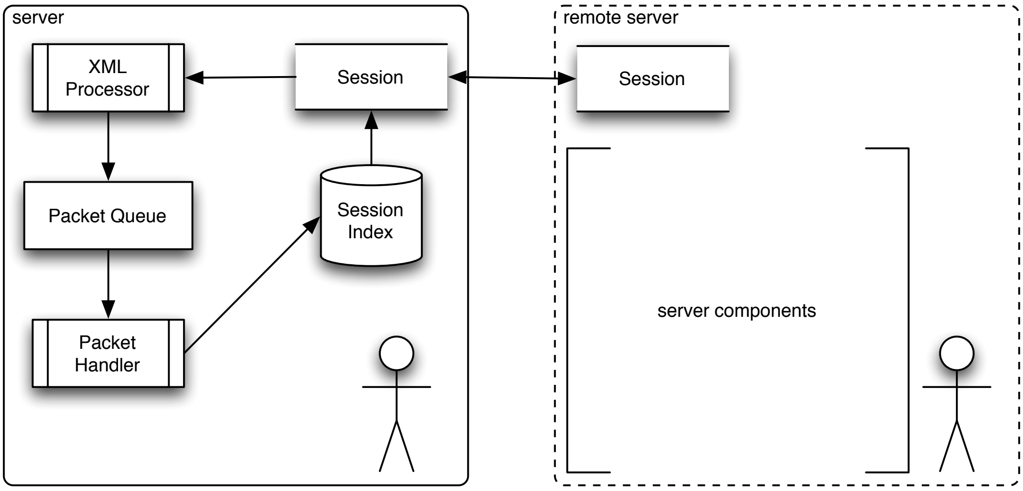


Figure 6.5: A simplified overview of two servers communicating.

3. XML parser – the XML parser takes packets and converts them in to Java objects for use in handler endpoints.

4. Handlers – after a packet has been parsed, it is sent to the registered handler for its type, which processes the packet. For example, a message handler would only take `<message>` packets and would send them to the correct GUI window for the sending entity.

Various threads will be created to allow for concurrent processing of multiple connections, however all these threads will run under a parent `Server` thread that maintains connections and sessions, along with dealing with registering packet handlers.

The server uses `java.net.ServerSocket` to create a socket that accepts all connections on a specified port. The port is defined during object construction, generally it should be random, and `Server` will throw an `IOException` if the port is already taken.

6.4.2.2 Session

The `Session` and `SessionIndex` classes form the basis around managing connections between servers – a `Session` object represents a single connection between two entities. `Session` objects store a number of bits of information regarding a connection that provide context, such as the connection ID (for use in sending messages to the correct GUI window), the `java.net.Socket` associated with the connection, the stream objects for sending/receiving data, statuses, and so on. A `Session` object is created upon connection and filled with the relevant data (random connection ID, user details, etc.) and as long as the connection remains open the `Session` object will remain in memory.

However, Java objects are transient and cannot be easily transmitted across a network, particularly using XML. The `Server` maintains an objects called `SessionIndex`, which is a wrapper around a `java.util.Hashtable` where the key is the connection ID (also referred to as a session ID) and the value is the corresponding `Session` object. By doing

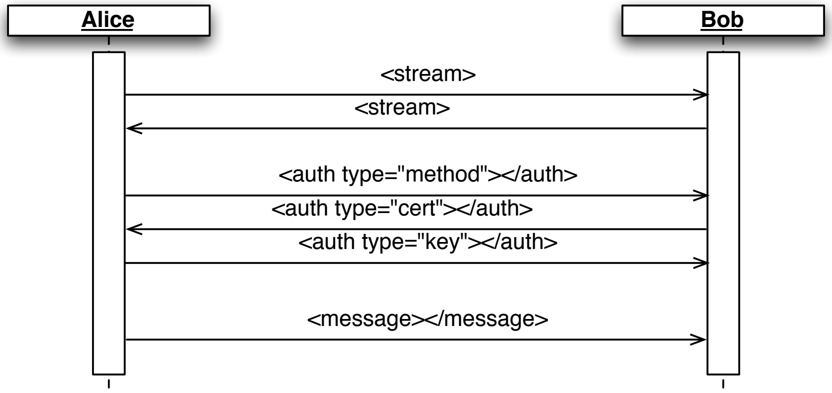


FIGURE 6.6: The basic conversation elements between two entities.

this, we are able to maintain a persistent store of information regarding a connection that is easily accessible from all objects within the `Server`, like packet handlers.

The `Session` is also used to store, upon connection, a `Conversation` object for the newly opened conversation window. Upon receipt of a `<message>`, the message handler will look up the session using the session ID received with the packet, and use methods within the `Conversation` object to update the window with the received message.

As local `Session` objects are created by the server when a outbound connection attempt is made, they are used to handle the assembly and transmission of packets, such as connection packets, message, authentication, etc.

6.4.3 Protocol Model

For structural information regarding the protocol, see §*B.*

The protocol model is based around using XML (eXtensible Markup Language) to transmit data along with metadata (attributes) to represent instructions to be received by another system.

6.4.3.1 Routing and How It Works

We will find out in this section how packets are routed through the server upon receipt, along with how they are sent. Figure 6.6 provides an overview of how two entities will communicate with one another, and what packets are required.

6.4.4 Parsing XML

Before we can start converting received messages in to Java objects and handling them, we must parse the XML in to something more usable.

6.4.4.1 SAX and Xerces

An implementation of the Simple API for XML (SAX) known as Apache Xerces is used as the SAX parser for incoming packets. Originally a Java-only library, it has become the standard for XML parsing within Java. It uses an event-driven model to parse XML documents, and rather than handling entire documents at once, it is done in a continuous stream of pieces, unlike DOM parsers (Document Object Model) which run using trees based on a complete document. The benefits of this for us are three-fold:

1. It is considerably faster than standard parsing, and works in linear time.
2. Only small portions of a document need be in memory at any one time, and so the memory footprint of parsing a document is small.
3. It can handle not being in possession of a full, valid XML document when parsing, like the packets we will be sending between entities.

It has one primary downside however: developer time. The concept of event-driven XML parsing is harder to manage than simply imagining a document as a tree. However, using Apache Xerces significantly improves upon this as the developer no longer has to handler the parsing itself, but merely what to do with the data once it is parsed. Xerces is more accurately a modular library for parsing, validating and manipulating XML documents. It introduces a framework known as *Xerces Native Interface* that allows developers to build their own parser components, however we will only be using it for its SAX features.

SAX parsers will follow through a document, and upon reaching specific points it will generate events and execute a callback function. For example, upon reaching the start of an element – e.g. `<message>` – the parser could be configured to call a function `startElement()` that determines what to do with this new element.

One interesting thing to note is the use of the class `StreamingCharFactory`. While capable of handling subsections of XML documents, SAX parsers are not generally (by default) able to handle streaming data as it arrives, and so the parsers will hang until reaching the end of the document (in this case, the connection closes). [27] details the `StreamingCharFactory` class that configures Xerces to use a streaming data reader rather than the default buffered one.

The InputHandler class deals with XML parsing by processing these SAX events. The primary way that documents will be handled is through *depth* – each section of the document is treated as a tree (as in DOM parsing). Upon encountering each element, the depth is increased by one, and decreased by one when the element ends. This is known as depth-first searching. We have three main methods that handle events:

1. `startElement([...])` – aptly named method that handles the start of elements.
2. `characters(char[] c,int start,int length)` – handles the string values of elements.
3. `endElement(String uri,String localName,String name)` – handles the closing tags for the current open element.

Methods 1 and 2 create `Packet` objects that represent their respective XML elements. In the case of `startElement()`, when at depth 0 the only element possible is a root `<stream>` and so a `Packet` with element name "stream" is created and pushed on to the queue, along with the attributes passed down from the SAX parser:

```java
public void startElement(String namespace,
                                String localName,
                                String name,
                                Attributes attributes)
                        throws SAXException {
  switch (depth++){
    case 0: // Root element
      if (name.equals("stream")){
        Packet packet = new Packet(null,name,namespace,attributes);
        packet.setSession(session);
        packet_queue.push(packet);
        return;
      }
      throw new SAXException("Root element must be <stream>");
    case 1: // Message elements
      packet = new Packet(null,name,namespace,attributes);
      packet.setSession(session);
      break;
    default: // Any child elements
      Packet child = new Packet(packet,name,namespace,attributes);
      packet = child;
  }
}

```

As can be seen, at depth 1 a `Packet` is created with all the provided attributes from the parser, including the name. At any other depth, however, we can assume we are dealing with a child element and so the previous `Packet` is set as its parent (see §6.4.4.2 for details about how parent-child relationships are implemented programmatically) and all other attributes are stored as usual. At each depth, a global `Packet` object is stored in the class to give access child packets access to parents.

If we are dealing with a value of an element, for example the body of a message, the method `characters()` is called by the parser. `characters()` is a simple method that adds a child value to the last traversed packet in the tree.

```java
public void characters(char[] c,int start,int length) throws SAXException {
  if (depth > 1) packet.getChildren().add(new String(c,start,length));
}

```

Finally, when an end element is reached we will once again check the current depth. If it is of depth 0, we must be at the end of the document and so a `</stream>` packet is created and pushed on to the queue. At depth 1, the current packet is complete and can be pushed on to the queue, and otherwise we must still be assembling the parent, and so set the pointer to the parent packet.

```java
public void endElement(String uri,
                            String localName,
                             String name)
       throws SAXException {
  switch(--depth){
    case 0: // End of the stream
      Packet c_packet = new Packet("/stream");
      c_packet.setSession(session);
      packet_queue.push(c_packet);
      break;
    case 1: // Put the completed packet on the packet queue
      packet_queue.push(packet);
      break;
    default:  // Parent still being constructed; traverse back up the tree
      packet = packet.getParent();
  }
}

```

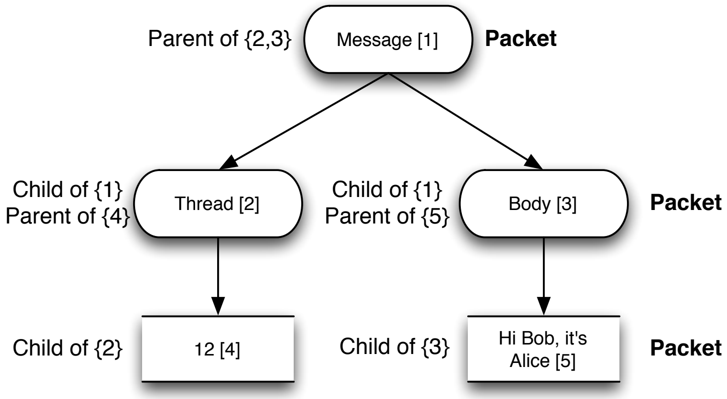
6.4.4.2 Packets

All packet related classes can be found in *com.cyanoryx.uni.enigma.protocol.xml.*

Packets in the context of the Enigma protocol are somewhat distinct from what are traditionally known as packets in networking. When we refer to packets we are referencing the XML sections representing messages sent between entities. In the previous section, §6.4.4.1, we defined how these XML fragments are parsed and converted in to packets without really defining what a packet in this context is (henceforth any references to "packet" will refer to the concept of sections of XML, unless stated otherwise or defined as a class, e.g. `Packet`). Previously we described creating packets by instantiating `Packet` objects with the name, attributes and child elements of the XML – this class serves two purposes:

1. To provide the capability of storing packets internally and in memory without the need to re-parse each time the data is needed, and to provide helper methods to retrieve this data.

2. To allow the simple, programmatic creation of XML packets through the instantiation of `Packet` objects which can then be converted in to XML to be transmitted across the network.



This graphic displays how Packet classes represent XML elements and store their children in what is effectively a tree. The corresponding XML would be:

```
<message>
  <thread>12</thread>
  <body>Hi Bob, it's Alice</body>
</message>
```

The Packet class can be simply defined overall as a data structure consisting of a name, and attributes, with a java.util.LinkedList of child Packets and a pointer to a parent Packet, where applicable.

Packets are parsed and handled on a first come, first served basis. Upon receipt, the Server places packets in to a PacketQueue where they will be picked out one by one by a running QueueThread and handled by an appropriate class. §6.4.5 provides an overview of how Packets, queues and thread fit together in the overall system.

Packet can be found in *com.cyanoryx.uni.enima.net.protocol.xml.Packet.*

6.4.4.3 Packet Handling

The PacketQueue class is a thread-safe (synchronized) class that acts as a wrapper around a java.util.LinkedList for storing all incoming packets. It has two methods, push and pull, the latter removes and returns the packet at the top of the queue, and the former adds a packet to the bottom of the queue. Both, alongside being implemented as synchronized, are thread-safe in that push notifies all threads awaiting use of the queue when it is finished add the current packet, and pull forces threads to wait if the queue is empty.

The PacketListener class, or more accurately the interface, provides a notify method implemented by the event handlers (§6.4.4.4) which allows the QueueThread (below) to notify said handlers when a packet is pulled from the queue.

The QueueThread class is perhaps the most important as it acts as the link between receiving a message and handling it. A QueueThread is started when a server is first created, and is passed all the desired event handlers and their XML tag identifiers, which are stored in a java.util.HashMap. For example, the OpenStreamHandler is required to handle new connection requests, signified by <stream> and as such its identifier is "stream." The QueueThread is constantly running alongside the server, looking for new packets pushed on to the queue. When it notices a new packet, it pulls it out of the queue and checks the element name of the packet and looks for it in its handler HashMap. If a handler is found, it is notified and passed the Packet object.

The idea for this came from [27].

```java
public void run(){
  for (Packet packet=packetQueue.pull();
       packet!=null;
       packet=packetQueue.pull()) {
    try {
      // Element name to look for.
      // Matches name of listener
      String match = packet.getElement();

      synchronized(packetListeners){
        Iterator<PacketListener> iter = packetListeners.keySet()
                                                       .iterator();
        // Loop through the current set of listeners
        while (iter.hasNext()){
          PacketListener listener = iter.next();
          // Get the name of the tag to match
          String listenerString = packetListeners.get(listener);
          // If the packet's element matches the
          // element for this listener...
          if (listenerString.equals(match)){
            listener.notify(packet); // ..send packet to handlers
          }
        }
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

`QueueThread` allows many packets to be sent to a server at once, with each being handled in order and none being lost.

The ProcessThread class is created whenever a new connection is made to the `Server`. As `InputHandler` objects parse an entire XML document, they are tied-up with each connection until the document is complete (i.e. a `</stream>` is sent). Because of this, if we were to use one `InputHandler` per server, only one connection would be possible at any one time. As a solution, `ProcessThread` is created for each connection, which instantiates a new `InputHandler` object and begins the XML processing on the incoming stream.

6.4.4.4 Handlers

Handler classes are where most of the work is done once a packet has been parsed. Four are required for the basic operation of the Enigma protocol:

1. **OpenStreamHandler** – handles an open stream request and initiates some default settings such as whether or not the conversation will be authenticated – `<stream>`.

2. **CloseStreamHandler** – gracefully handles closed connections and cleans up `Session` objects – `</stream>`.

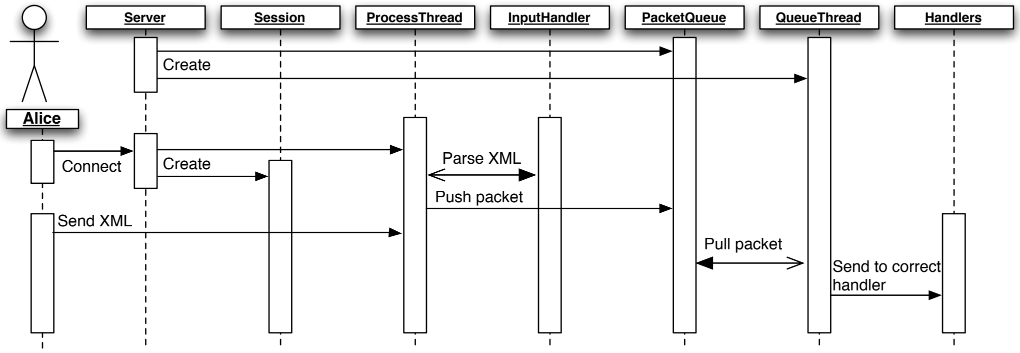
3. **AuthHandler** – handles all authentication and encryption setup, including received public keys, session keys and so on – `<auth></auth>`.
4. **MessageHandler** – handles all incoming messages, determines who they are from and displays them in the appropriate window – `<message></message>`.

Handlers implement the `PacketListener` interface, meaning they are notified whenever the matching packet type is parsed out of the queue. See §6.4.4.3 for further details about how `QueueThread` determines which handler to use. Handlers, on creation, also receive a reference to the master `SessionIndex` within `Server`.

The handler implementations can be found at *com.cyanoryx.uni.enigma.server*.

6.4.5 Fitting the Protocol Components Together

So, how do these classes fit together in the final system? We've so far given distinct diagrams, and provided abstract descriptions of classes and how they relate, but this is hard to conceive as a whole system.



This figure shows the process of `Server` creation, through to client connection and and message receipt.

6.5 Algorithm Implementation

*Packages covered in this section: com.cyanoryx.uni.crypto.**

6.5.1 Interfaces and Abstraction

One of the major points of the Enigma application is the ability to easily introduce new symmetric or asymmetric algorithms in to the application. To do this, we use a well known concept of object-oriented programming: abstraction. Abstraction, in its basic form, is the process of representing some data or logic but hiding the actual implementation. A good example of this is our `LogHandler` earlier, which provides an abstraction of outputting log information – the developer/program is provided with an interface with, for example, a `log()` method that displays a supplied string in a log window. The implementation of the log function is irrelevant to the developer, all they need to know is that they can use it to reliably log information. Because of this, the logic behind logging data can be changed as necessary assuming it still has the same output – this is known as a "black box."

In Java, abstraction is implemented using *interfaces*. An interface defines methods along with their input and output data types, and any class that *implements* this interface must create the logic for these methods. How are interfaces applicable to Enigma? Producing a program that encrypts data using multiple libraries can be difficult as each may have subtle differences in how they expect to receive and output data. For example, a block cipher A may implement a `encrypt()` method that takes portions of data as byte arrays, whereas a cipher B may take the entire plaintext as `String`. To solve this, an interface will be introduced with an `encrypt` method that requires all implementing classes use it – this results in a standard implementation of all algorithms, meaning a developer knows without a doubt what to input and what to expect back.

This is particularly applicable to Enigma as it means code repetition when dynamically switching between algorithms is reduced significantly. Alongside this, if a new algorithm is desired for use, it simply needs to implement the Enigma encryption interface.

6.5.2 Key Agreement and Certficates

The initiator of a conversation sends their certificate (including their public-key signed by a Certificate Authority) immediately after a successful connection is made. If the certificate is valid and verifiable, the receiver will generate a session key and encrypt it using the initiator's public key before transmitted it. Both users now have a shared session key, and can begin encrypting messages before sending.

This is the standard flow for authentication and key agreement. This takes place primarily in `Connect#connect()` and `AuthHandler#notify()`, which can be found in *com.cyanoryx.uni.enigma.gui* and *com.cyanoryx.uni.enigma.net.server*, respectively.

Key Generation is performed through an option in the toolbar – Options > Generate Keys... – and requires no user input other than having previously set the Certificate Authority's key location in the preferences.

This utility automatically generates a public/private key pair for the user and then signs it with the Certificate Authority's private key to produce their own certificate which will be used by other entities to confirm their identities. In the Real World, users would not have access to the private keys of the CA, however as this application is for research and will not utilise an actual CA we will assume the ownership of CA private keys.

6.5.3 Ciphers

Now that we have a shared session key, messages can now be encrypted by either entity and transmitted over the network securely.

Sending a message is handled by `Session#sendMessage()` which will, before sending a message, check to see if the current connection requires encrypted messages and if so it encrypts them using the chosen algorithm. For example, AES:

```
AES aes = new AES();

Key k = new Key(KeySize.K256);
k.setKey(key);
aes.setKey(k);

aes.setPlainText(body.getBytes());
msg=Base64.encodeBytes(aes.encrypt());
```

As can be seen, messages are encoded using Base64 before transmission. Symmetric ciphers encrypt in blocks of bytes, and so trying to send the output from a cipher directly will result in sending binary data which can cause issues within XML documents. Base64 is an encoding scheme that takes binary data and represents it as ASCII text, meaning it can be printed and displayed using almost all character sets. The inner workings of Base64 are out of the scope of this project, however §D lists the open source library used for encoding and decoding.

It should be noted that the actual XML of a message is not encrypted, but only the body.

Receiving a message is simple, first we must check to see if we are expecting and allowing encrypted messages, and if so create the appropriate cipher object. For example, using AES:

```
AES aes = new AES();
Key k = new Key(KeySize.K256);
// Get the session key from the Session object
k.setKey(s.getCipherKey());
aes.setKey(k);
// Convert the Base64 encoded message to binary
aes.setCipherText(Base64.decode(message));
// And decrypt it, producing a String object from the
// byte array
message = new String(aes.decrypt());
```

The message will then, as usual, be sent to the appropriate conversation window.

6.6 Testing

6.6.1 Test Driven Development

A fairly new (or at least recently popular) development methodology was used for the application and algorithms development: test driven development. After creating a basic set of requirements for each unit of code, a test is written for that unit that defines what the input and output should be. Each line of code written should be a response to a part of a test.

Software engineering is often compared to other engineering disciplines, like those that involve bridge building. However these metaphors tend to fall apart as you go deeper. Testing in non-software related engineering fields is difficult – you have to build the product to test it – whereas it is easy, even cheap, in software engineering to create tests *before* the product has been built and run them until they pass.

Regularly developers, and even managers, will claim that there is not enough time to test or that it is too much overhead, but if we are to use metaphors anyway we can use them for this: an engineer creating a building cannot simply put off tests because they don't have time, otherwise they endanger lives. Software engineering errors can endanger lives[28], and testing reduces that risk considerably. *Testing is the engineering rigour of software development*[29].

6.6.2 Unit Testing

Being the primary, basic form of testing that most developers use for requirements testing, unit tests are a form of black box testing. This means that all that is taken in to account is the input to and output from the program, thus treating the unit of code as a *black box* where the inner workings cannot be determined.

Unit tests will be our main method for testing as they are simple and quick to implement, reducing the overhead and supposed time-sink of creating tests.

Each root package in the `com.cyanoryx.uni` tree contains a *test* directory with the appropriate unit tests.

6.6.3 Code Coverage

Code coverage (CC) is a method of measuring how many lines of code in a project are executed during automated testing. It is a useful tool for determining what percentage of an application is covered by tests, meaning that we are able to say whether or not said application is provably working as expected by computing if our passing tests represent the program as a whole.

There are many utilities for calculating this, and in our case an open-source toolkit known as *EMMA*[2] was used.

6.6.4 Functional Testing

Functional testing is a subset of black box testing that involves (mostly automatically) testing that user interface components work properly and buttons do what they're designed to do. However, it was decided in the case of Enigma that due to the low complexity of the software and use-flow, functional testing could be done manually to ensure that the software worked as properly.

[2] http://emma.sourceforge.net/

6.7 Summary

We have now listed all the general components that make up the Enigma application: interface, server and algorithms. We will now go on to think about the algorithms and their implementations in further detail – known as cryptanalysis.

6.8 Usage

A user's instruction manual is included as §*D*. This manual also briefly covers compilation, required dependencies, and other build related information.

6.9 Program Listing

Due to the size of the project a detailed, printed code listing is impossible, and thus it is recommended that the code be viewed using a text editor or other text environment on a device. If you do not have a digital copy of this project, please see §*1.4.2*.

Chapter 7

Outcomes

7.1 Fulfilment of Specification

All points in the specification (see §A) were met fully, and the requirements have been fulfilled entirely. The primary purpose has been fulfilled: to create an application that will allow the testing of the efficacy of cryptographic algorithms within a real-life application.

7.2 Comparison to JDK

7.2.1 RSA

Library	Average Time (20 iterations; nanoseconds)
Enigma	6662350
JDK	445450
Difference	6216900

7.2.2 Symmetric Cryptography (AES)

Library	Average Time (20 iterations; nanoseconds)
Enigma	293350
JDK	60550
Difference	232800

7.2.3 Asessment

As can be seen, the Enigma versions of both algorithms are considerably slower than the JDK versions – indeed, in both cases an *order of magnitude* slower. Why is this the case?

1. There are many aspects where the algorithms could differ, such as in implementations of minor details like Mask Generation Functions, or hashing functions.
2. Large amounts of time has been devoted to the Java SDK encryption library, meaning that it is extremely mature and any efficiency issues have long been eliminated.

In reality, the author cannot compete with the manpower available to the larger open source projects, which leads neatly in to our summary.

7.3 Summary

Our aim in this project was to produce, from scratch and using only technical documentation, implementations of cryptographic algorithms that could be used in a real world application. This objective was completed successfully, and a large amount of new knowledge has been gained. However, the final recommendation? Do not do this. Cryptographic implementations have long been standardised and produced in mature libraries that are easy to use.

The main point is this: if you are doing something so unusual with cryptography such that you need to implement algorithms yourself, then you shouldn't be doing that. If you aren't doing something unusual, then you can use the libraries already available.

This brings us back to the quote from Bruce Schneier given at the beginning of this report:

"Anyone can design a security system that he cannot break. So when someone announces, "Heres my security system, and I can't break it, your first reaction should be, Who are you? If he's someone who has broken dozens of similar systems, his system is worth looking at. If he's never broken anything, the chance is zero that it will be any good."

Non-standard and custom implementations are only truly safe when they have been created by developers experienced in the production of cryptosystems and tested thoroughly.

The author hopes that this report has been of interest to you, or at least beneficial to your knowledge of information security, cryptosystems and how difficult it really is to get it right.

Appendix A

Enigma Application Software Requirements Specification

A.1 Introduction

This document lists basic requirements for the Enigma application. All these conditions must be met for the proper working of the application and for it to work as expected. This is not a high-level design specification, but an overview.

A.1.1 Scope

This document applies to the overall Enigma application that will function as an instant-messaging server. The individual cryptographic algorithms are not specified here.

A.2 Overall Description

The purpose of the Enigma application is to provide a test bed for public-key and asymmetric encryption algorithms so that they can be tested in a real-world use environment and evaluate performance using external tools.

A.2.1 Product Functions

A.2.1.1 Primary Functions

- To connect to and handshake with a remote Enigma server.
- To receive and accept connection requests from remote Enigma servers.
- To send ASCII text messages between two entities connected through Enigma servers.
- To be able to “toggle” encryption of messages on/off, allowing for secure and insecure sessions.

- To use public-key cryptography to securely agree on a shared session key for use in symmetric ciphers.
- To securely encrypt and decrypt messages given a designated cipher function and a key.
- To verify the identity of a remote entity using a public-key certificate.

A.2.2 Interfaces and Accessibility

The application will have a graphical user interface (GUI) that will use the native components of the host operating system.

A.2.3 User Characteristics

The user will likely be:

- A developer or someone who is knowledgeable about computer systems.
- A person who is connected to a developer and is aiding in testing/use for analysis, and thus has instructions for use.

A.2.4 Constraints

The software should **not** be considered for use in an environment where sensitive information is required to be shared which, if released, would pose a threat to life, cause injury or damage to persons or property, or any other result which could be considered damaging or illegal.

No guarantees of running-time can be made, and neither the encryption, decryption or transmission of data should be considered real-time.

The software must be used purely for research purposes.

A.2.5 Assumptions and dependencies

- The host computer(s) are capable of running at least Java 6.
- The source must be compiled with the appropriate library dependencies.
- Two entities wishing to communicate must be connected via a network, be it a local connection or wider (such as the internet).

Appendix B

Enigma Protocol Specification

B.1 Introduction

The Enigma Protocol is a communications protocol intended for use in Instant Messaging applications. It is very loosely based around the concept of *Jabber/XMPP*[1], primarily as it is an application of the Extensible Markup Language (XML) to allow for near-real-time communications between networked devices. The XMPP protocol is considerably more complex than Enigma, and the two should not be considered comparable.

This document covers the basic XML elements that must be implemented by an Enigma Protocol-based application. As its primary purpose is to provide a simple way of sending text with meta-data, it should only be used for research purposes. The Enigma Application can be considered a reference implementation, however it is not a complete implementation.

B.2 Requirements

An IM application for use in the testing of sending encrypted messages should have the capability to:

1. Be able to exchange brief text messages in near-real-time.
2. Transmit information that can be used to securely generate and exchange encryption keys.
3. Transmit information that can be used to identify and authenticate.

And thus, any application implementing the Enigma Protocol must be able to adhere to the above requirements.

[1] http://xmpp.org/about-xmpp/history/

B.3 History

This document covers version 1.0 of the Enigma protocol. There are no prior implementations.

B.4 Terminology

No specialised terminology is used without explanation in this document.

B.5 Format

A conversation between two users should be considered as the building of a valid XML document using the `enigma:client` namespace. The document should consist of a valid root element occurring only once, valid child elements matching those listed in the document, and finish with a closing tag matching the root element. Each packet, that is not a root element, must consist of a start-tag and end-tag, or an empty-element tag otherwise it **should not** be parsed and **should** be dropped.

The order of the messages is important if the `time` attribute is not included, in which case they should be handled and displayed in a first-in-first-out order.

B.6 Commands

B.6.1 Connection

A connection is represented by a streaming XML document, with the root element being `<stream>`.

- **Opening a connection:**

```
<stream  to="SERVER_NAME"
         from="USER_NAME"
         id="SESSION_ID"
         return-port="LOCALHOST_INBOUND_PORT"
         xmlns="enigma:client">
```

- **Closing a connection:**

```
</stream>
```

B.6.2 Authentication

- **Toggling encryption:**

  ```
  <auth stage="streaming"
        id="SESSION_ID"
        type="toggle">
        [off|on]
  </auth>
  ```

  Note: this requires the agreement of both users. The connection should be closed if one user disagrees to change the current encryption status.

- **Asserting the key agreement method:**

  ```
  <auth stage="agreement"
        id="SESSION_ID"
        type="method">
        [method type identifier]
  </auth>
  ```

- **Publishing a certificate:**

  ```
  <auth stage="agreement"
        id="SESSION_ID"
        type="cert">
        [Base64 encoded Enigma Certificate]
  </auth>
  ```

- **Publish an encrypted symmetric cipher key:**

  ```
  <auth stage="agreement"
        id="SESSION_ID"
        type="key"
        [method="CIPHER_ALGORITHM"]>
        [Base64 encoded encrypted key]
  </auth>
  ```

B.6.3 Messaging

- **Sending a message:**

  ```
  <message  [to="REMOTE_USER_NAME"]
            from="USER_NAME"
            id="SESSION_ID"
            [type=""]>
      <body>MESSAGE_CONTENT</body>
  </message>
  ```

B.6.4 Errors

Errors do not have a specific element themselves, but are included as a subelement of any other tag, setting `att:type` to `error`.

- **Sending an error:**

```
<element  [other attributes]
          type="error">
    <error type="ERROR_NUMBER">
        ERROR_MESSAGE
    </error>
    [element contents]
</element>
```

Appendix C

Licenced Software Usage

The Enigma application and associated utilities make use of several libraries and projects, all of which are used legally and in accordance with the matching software licence. The individual licence declarations are included within the files, or where appropriate. Below each library/project is listed, with their locations and other details.

1. **Base64 Encoder/Decoder**
   Author: Robert Harder
   Website: http://www.iharder.net/current/java/base64/
   Licence: None. Public domain release, full usage allowed.
   Location: *com.cyanoryx.uni.common.Base64*

2. **Cryptix Byte Utilities**
   Author: The Cryptix Foundation Ltd.
   Website: http://www.cryptix.org/
   Licence: Cryptix General Licence - http://www.cryptix.org/LICENSE.TXT
   Location: *com.cyanoryx.uni.common.Bytes*
   Notes: Most methods only used partially.

3. **Apache Xerces (XML parsing)**
   Author: Apache Software Foundation
   Website: http://xerces.apache.org/
   Licence: Apache Licence, v2.0 - See licence at *dep/APACHE-LICENSE-2.0.txt*
   Location: *dep/xerces.jar*

4. **Instant Messaging in Java**
   Author: Iain Shigeoka (Manning)
   Website: http://www.manning.com/shigeoka/
   Licence: The Shigeoka Software Licence - See licence at *dep/SHIGEOKA-LICENCE.txt*
   Location: *com.cyanoryx.uni.enigma.net.io.XercesReader.java*, plus occasional usage of code fragments and methods, which are noted within the code.

Appendix D

Enigma: User Manual

D.1 Introduction

The Enigma application is designed to be relatively simple to use, primarily in that not much specific knowledge about the cryptosystems powering it is necessary. No specific details for each algorithm is required, and you are able to simply select whether or not you want encrypted conversations. The following instructions will get you started with the initial application start and setup, along with some tips on keeping your conversations secret.

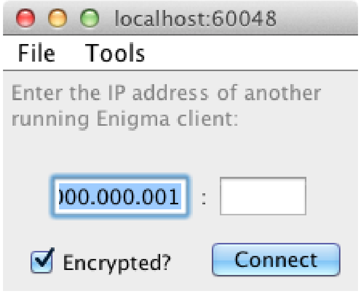
D.2 Installation

No installation is required, the application is self-contained in a Java `jar` archive – usually `Enigma.jar`. Some operating systems allow you to open the `jar` directly, however if that is not the case then from a command line shell running the directory where the Enigma application is stored, run:

```
$ java -jar Enigma.jar
```

D.3 Connecting and Sending Messages

The creation of a conversation is reliant on two entities separately running an Enigma server: as long as the Enigma application is running, a server is running and waiting for connections.



Enter the IP address of another running Enigma client:

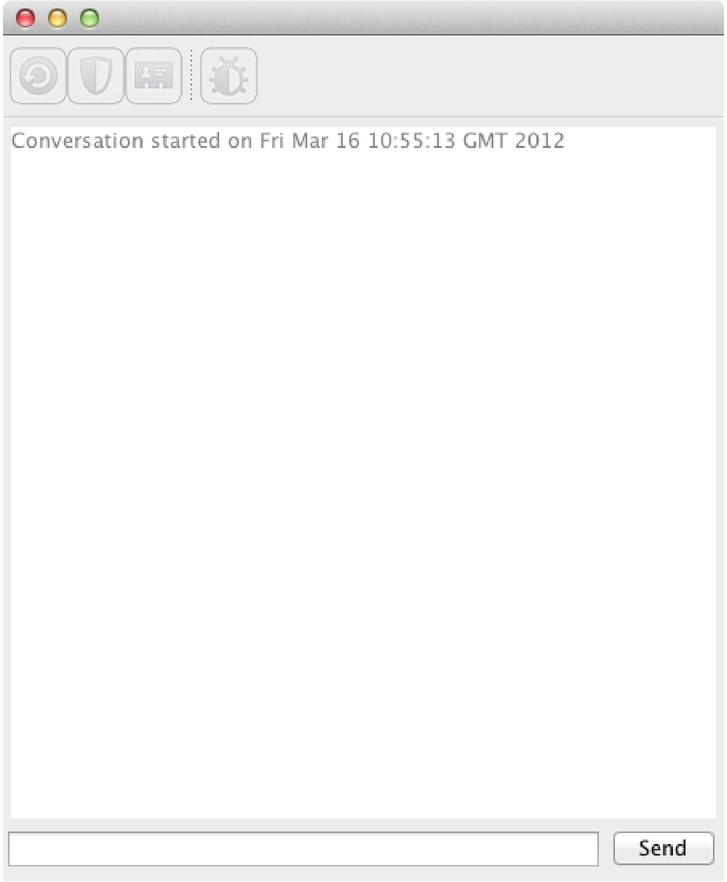
)00.000.001 :

☑ Encrypted?

The title bar of the window provides an important piece of information: the external port on which your server is running – other users will need this to contact you. The port is initially random in the range 60000 – 60100, however a default, fixed port can be set in Tools > Options > General. You do not need to do anything further to start receiving conversation requests – other users may try and open a conversation with you once they know your IP address and port number.

The same requirements are needed for you to converse with another user – their IP address and port number are needed, which are entered in to the input boxes in the main window. You will, however, need your key pairs to be generated first. See §D.5.1.

Upon connecting to a remote user or receiving a connection, a conversation window will automatically be opened.



Conversation started on Fri Mar 16 10:55:13 GMT 2012

D.4 Preferences

The preferences window can be access through the Tools menu: Tools > Options. The preferences save on change, and as such you do not need to manually click a "save" or "apply" button.

D.4.1 General

A locale can be set, assuming the appropriate language file has been made available locally. Enigma makes use of `.properties` preference files to store key/value representations of language strings that can occur within the application. See `Enigma.properties` for the keys required and their associated English strings.

The default port can be set and so all future servers started will be run on this port.

D.4.2 Personal

A personal name can be set which will appear to users conversing with you.

D.4.3 Security

D.4.3.1 General

Unauthenticated (and thus unencrypted) conversations are possible, however by default this option is off. Check the "Allow Unauthenticated Conversations?" checkbox to enable this. If a user tries to connect to you without authentication when this option is on, the conversation will not be accepted.

Certificate validation can be turned off for testing purposes, however it is not recommended. With this setting off, man-in-the-middle attacks are possible.

D.4.3.2 Key Agreement

The default agreement method defines the algorithm used to share keys securely between you and the remote user. This can be changed with no effect on the security of the application.

The certificate location must be specified after generation (if not done so automatically by the application) so that they can be shared with remote users.

The private key location must be specified (if not done so automatically by the application), otherwise authenticated conversations will not be possible.

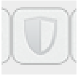


Figure D.1: The toggle encryption button

Certificate Authority keys (both public and private) are required as the Enigma application does not make use of actual Certificate Authorities. This setting requires the location of the public key with the extension `.pub` and assumes that the private key is at the same location minus that extension.

D.4.3.3 Cipher

The symmetric cipher to be used for encryption can be set, which will also have no effect on the security of conversations.

D.5 Cryptosystems

D.5.1 Generating Keys

Generating keys is simple and most of the work is done for you. First ensure that the location of the bundled Certificate Authority keys (public) is set in the preferences. Select "Generate Keys..." from the Tools menu (Tools > Generate Keys...) and a keypair and a certificate are automatically generated and their locations set in the preferences. No further configuration is required.

Symmetric cipher keys are generated randomly for each session and cannot be manually selected through the user interface.

D.5.1.1 Notes on Maintaining Secrecy

Your private key is the very basis of the security behind sharing a cipher key and thus maintaining secrecy. If your private key is believed to be lost or stolen then a new keypair should be generated immediately.

D.5.2 Encrypting Conversations

Conversations are set to be encrypted by default, and the application will handle this assuming you have setup your key pairs. Unencrypted conversations may be started by either user by unchecking the "Encrypted?" option, however both users must have the "Allow unauthenticated conversations?" option checked if an unencrypted conversation is desired.

Throughout a conversation, assuming the "Allow unauthenticated conversations?" preferences is checked, either user may toggle encryption on and off using the toggle button in the conversation window (see Figure D.1).

FIGURE D.2: The session key regeneration button

If you are currently in an encrypted conversation, you are using randomly generate session keys that are automatically shared between you and the remote user. For any reason, you are able to have the session keys regenerated and sent to the remote user. See Figure D.2.

D.6 Troubleshooting

D.6.1 Connection

Upon connection, the user may receive a message ”Could not connect” along with a brief description of the error.

Message	**Solution**
Port out of range	The given port number is invalid (must be less than 65535).
Connection refused	The IP address and port do not lead to a valid Enigma server.
Operation timed out	The server is not accepting packets on this port.
Can’t find bundle for base name Enigma, locale <locale ID>	The application could not find the `properties` file for your locale. Ensure that it is the same directory as the `jar` file.

D.6.2 Messages

The protocol specifies no way to acknowledge if a message has been received, and as such if the user believes that messages are not sending they should check the local logs for the application or the console (see §D.6.4).

D.6.3 Key Generation

Message	**Solution**
Could not find CA keys.	The Certificate Authority public/private key pair could not be found. Ensure that the location is set in Tools > Options > Security > Key Agreement.
Could not generate keys.	The keys are saved to the directory `.` (the current working directory). Ensure that this directory is writeable, and that the CA keys are available.

For further information if this does not help, see §D.6.4 as a stack trace will have been generated for any of the errors that may occur here.

If all else fails, the `com.cyanoryx.uni.crypto.rsa.GenerateKeys` class can be compiled and executed to produce a key pair.

D.6.4 Further Details

If you launched the application through a command line shell, you will be able to view any stack traces that are printed when an error occurs. For example, in the case of a file not being found and a useful error message not being generated in-application the stack trace is a good place to start on finding out what went wrong.

Bibliography

[1] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2nd edition, 2008.

[2] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 6th edition, 2007.

[3] Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography (Discrete Mathematics and Its Applications)*. CRC Press, 5th edition, 1996.

[4] Dec 2011. URL `http://davidkendal.net/articles/2011/12/lehmann-primality-test`.

[5] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes in p. Technical report, Indian Institute of Technology, Kanpur-208016, INDIA, 2002.

[6] Robert G. Salembier and Paul Southerington. An implementation of the aks primality test. Technical report, George Mason University, 2005.

[7] Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. Technical report, Institute for Theoretical Computer Science, ETH Zurich, 1994.

[8] Mathias Romme Shwarz and Soren Gronnegaard Andersen. A study of maurer's algorithm for finding provable primes in relation to the miller-rabin algorithm. Technical report, 2007.

[9] 2006. URL `http://developer.classpath.org/doc/java/math/BigInteger-source.html`.

[10] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[11] NIST. Fips 140-2 – security requirements for cryptographic modules. Technical report, Information Technology Laboratory, NIST, 2001.

[12] D Eastlake, S Crocker, and J Schiller. Randomness recommendations for security. Technical report, Network Working Group, 1994.

[13] Ronald L. Rivest and Robert D. Silverman. Are 'strong' primes needed for rsa? Technical report, Massachusetts Institute of Technology, 1999.

[14] John A. Gordon. Strong primes are easy to find. In *Proceedings of Eurocrypt*, 1985.

[15] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption - how to encrypt with rsa. Technical report, University of Calinfornia, 1995.

[16] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) No. 1: RSA Specifications Version 2.1.* RSA Laboratories, 2003.

[17] Wenbo Mao. *Modern Cryptography: Theory and Practice.* Prentice Hall, 1st edition, 2003.

[18] E. Rescorla. *Diffie-Hellman Key Agreement Method*, 1999.

[19] Brian Gladman. A specification for rijndael, the aes algorithm. Technical report, 2007.

[20] Federal Information Processing Standards. *Advanced Encryption Standard FIPS-197.* Federal Information Processing Standards, November 2001.

[21] 2005. URL `http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf`.

[22] B. Kaliski. Pkcs 5: Password-based cryptography specification. Technical report, RSA Laboratories, 2000.

[23] Neal R. Wagner. *The Laws of Cryptography.* 2nd edition, 2003.

[24] D. Cooper. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* NIST, 2008.

[25] S. Kent. *Privacy Enhancement for Internet Electronic Mail: Part II.* IETF, 1993.

[26] Steve McConnell. *Code Complete.* Microsoft Press, 2nd edition, 2004.

[27] Ian Shigeoka. *Instant Messaging in Java.* Manning, 2002.

[28] Nancy Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[29] Kevlin Henney, editor. *97 Things Every Programmer Should Know.* O'Reilly, 2010.