School of Engineering, Computer, and Mathematical Sciences

Highly Secure Systems

COMP716

2018 Semester 2

Highly Secure Systems - COMP716
Course Manual
Auckland University Of Technology

This project is on GitHub, find it
and download the source files at:
https://github.com/millecodex/COMP716

Version 3.0 by Jeff Nijsse, 2018.
Jeff.Nijsse@aut.ac.nz
Version 2.0 by Seth Hall, 2015.
Version 1.0 by Andrew Ensor, 2006.

Contents

Chapter 1

Symmetric Ciphers

1.1 Classical Encryption Techniques

An *encryption* algorithm is an algorithm which takes a message, called *plaintext*, and converts it into a coded message, called *ciphertext*, that is not readily intelligible without special knowledge. For encryption to be useful it must be possible to reverse the algorithm, decrypting the ciphertext and obtaining the original plaintext. The study of encryption is known as *cryptography*, whereas *cryptanalysis* uses techniques to decipher ciphertext without being provided with full knowledge of the enciphering details, thus *breaking* the code and obtaining the original plaintext.

Encryption algorithms are usually built using combinations of two simple techniques:

substitution where elements in the plaintext are replaced (substituted) with other elements,

transposition where elements in the plaintext are rearranged (transposed) with each other.

An encryption algorithm makes use of a *key* that helps determine the resulting ciphertext, with the aim of making it difficult to decipher ciphertext without prior knowledge of the correct key.

One of the earliest and simplest substitution ciphers is known as the *Caesar cipher* as it was used by Julius Caesar during military campaigns. Each letter of plaintext is replaced by a letter some fixed number of positions further down the alphabet, wrapping around beyond the letter `z` back to the letter `a`. For example, using a key that gives a rotation of three places and denoting plaintext in lowercase and ciphertext in uppercase gives:

Plaintext: `secure and reliable systems`
Ciphertext: `VHFXUH DQG UHOLDEOH VBVWHPV`

where spaces have been included for clarity. Decryption is achieved simply by rotating each letter of the ciphertext back three places. Note that it is essential for the security of the Caesar cipher that this algorithm be unknown to anyone intercepting the ciphertext, as by brute-force the ciphertext could then be decrypted by trying all 26 possible cases of a key, presuming the original plaintext would be recognizable when found.

A *monoalphabetic substitution cipher* generalizes and improves on the Caesar cipher by using substitutions determined by a random permutation of the letters of the alphabet, replacing the letter `a` by the first letter in the permutation, `b` by the second, and so forth. For example, using *`qwertyuiopasdfghjklzxcvbnm`* as the key, so that the letter `a` is substituted by `Q` and `s` by `L` gives:

Plaintext: `secure and reliable systems`
Ciphertext: `LTEXKT QFR KTSOQWST LNLZTDL`

(in practice spaces too would be encoded if included as part of the plaintext). A brute-force approach would require checking on average half of the $26! \approx 4 \times 10^{26}$ possible keys, which is totally infeasible. However, like any simple substitution cipher this cipher has a fatal flaw in that it preserves much of the structure of the plaintext. If the type of information in the plaintext is known, such as English text, then a *frequency analysis* can be performed on the elements of

the ciphertext, comparing the frequencies of each ciphertext element with expected frequencies for that type of text. For example, if a cryptanalyst suspects that `LTEXKTQFRKTSOQWSTLNLZTDL` is encoded from English plaintext then the fact that `T` occurs the most times in the ciphertext suggests that it is probably the ciphertext for the letter `e`, and the other frequencies give further insight into the key. Provided with a longer ciphertext or multiple ciphertexts encrypted with the same key a cryptanalyst has no problems in breaking the code. If parts of the plaintext are known (for example, many file formats such as PDF start with a well-known fixed header), known to contains a particular sequence repeated (such as `"the"`), or worse still if a cryptanalyst has access to a sample of both plaintext and its corresponding ciphertext, the code can be easily broken.

Relative Frequency of Letters in English Text

Letter	*Frequency*	*Letter*	*Frequency*	*Letter*	*Frequency*
a	8.167%	b	1.492%	c	2.782%
d	4.253%	e	12.702%	f	2.228%
g	2.015%	h	6.094%	i	6.996%
j	0.153%	k	0.772%	l	4.025%
m	2.406%	n	6.749%	o	7.507%
p	1.929%	q	0.095%	r	5.987%
s	6.327%	t	9.056%	u	2.758%
v	0.978%	w	2.360%	x	0.150%
y	1.974%	z	0.074%		

One attempt to improve on the monoalphabetic substitution cipher is to assign multiple ciphertext keys to common letters such as `e` to ensure the ciphertext does not contain any particular letters with noticeable frequency. However, this is still susceptible to a frequency analysis since certain combinations of pairs of letters, called *bigrams*, occur more frequently than others. Given a suitably large ciphertext the code could still be successfully broken using a frequency analysis of bigrams.

Relative Frequency of Bigrams in English Text

Bigram	*Frequency*	*Bigram*	*Frequency*	*Bigram*	*Frequency*
th	3.883%	he	3.681%	in	2.284%
er	2.178%	an	2.140%	re	1.749%
nd	1.572%	on	1.418%	en	1.383%

A *Hill cipher* encrypts blocks of m elements at a time using an $m \times m$ matrix $K = (k_{ij})$ as the key. A plaintext message is broken into blocks each with m elements (possibly requiring the end of the plaintext to be padded so its length is a multiple of m), then each block $p_1 p_2 \dots p_m$ is encoded to a block $c_1 c_2 \dots c_m$ of ciphertext using matrix multiplication modulo 26:

$$\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} k_{11} & k_{12} & \cdots & k_{1m} \\ k_{21} & k_{22} & \cdots & k_{2m} \\ \vdots & \vdots & & \vdots \\ k_{m1} & k_{m2} & \cdots & k_{mm} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix} \mod 26$$

where each letter is assigned a numerical value, `a` as 0, `b` as 1, ..., `z` as 25, and multiplication and addition are performed modulo 26. The matrix K must be invertible (modulo 26) to ensure that the ciphertext can be reversed using the inverse matrix K^{-1}.

As an example, the matrix $K = \begin{pmatrix} 8 & 21 & 21 \\ 5 & 8 & 12 \\ 10 & 21 & 8 \end{pmatrix}$ can be used to encrypt the plaintext `secureandreliablesystems` three characters at a time, where the first block `sec` would be encoded by:

$$\begin{pmatrix} 8 & 21 & 21 \\ 5 & 8 & 12 \\ 10 & 21 & 8 \end{pmatrix} \begin{pmatrix} 18 \\ 4 \\ 2 \end{pmatrix} \mod 26 = \begin{pmatrix} 10 \\ 16 \\ 20 \end{pmatrix}$$

corresponding to the ciphertext `KQU`. The entire plaintext would be encoded as `KQU DYR YKL JPE HAK ERA HYQ MUU`. As a 3×3 matrix has been used it is not susceptible to a frequency analysis of bigrams, and the larger the size of matrix the more of the structure of the plaintext that is hidden. However, the Hill cipher is particularly vulnerable to a *plaintext attack*, where some plaintext and its corresponding ciphertext are known. Possibly with just as few as m blocks of plaintext and its ciphertext the entries in the matrix K could be determined by solving a system of m linear equations, and so too the inverse matrix K^{-1} found for decryption.

The *Vigenère cipher* was originally described in 1553 and is an example of a *polyalphabetic substitution cipher*, where there are several monoalphabetic substitution ciphers available and the one chosen for each element of plaintext depends on the key. The Vigenère cipher simply uses the 26 possible Caesar ciphers and cycles through the letters of a key to determine which rotation to use for each element. For example, encrypting `secureandreliablesystems` using the key *`emily`* gives:

Key:	*`emilye mil yemilyem ilyemil`*
Plaintext:	`secure and reliable systems`
Ciphertext:	`WQKFPI MVO PIXQLZPQ AJQXQUD`

Note that one strength of the Vigenère cipher is that a frequent letter such as `e` gets encoded to different ciphertext elements depending on its position in the plaintext. The first successful cryptanalyst attack on the Vigenère cipher was made in 1854 by Charles Babbage, but is known as the *Kasiski examination*. It works by first attempting to determine the length m of the key. Since occurrences of common words such as `"the"` have a $\frac{1}{m}$ chance of being encrypted using the same key letters, if the plaintext is many times longer than the key then repeated patterns will result in the ciphertext (such as the repeated pattern `PI` in the above ciphertext example). These repeated patterns would occur at multiples of the key length, allowing guesses at the probable value of m. Once m is determined the Vigenère cipher is quickly broken as the ciphertext is next split into m sections, one per letter of the key and up to 26 brute-force attacks or a frequency analysis used to break each of the m separate Caesar ciphers.

The *one-time pad* is a modification of the Vigenère cipher where a random key is used that is at least as long as the message to be transmitted, and then the key gets discarded. Although this can be shown to give a theoretically secure cipher, for most applications it is not practical to first distribute a secret key for each individual message.

Instead, *rotor machines* such as the famous German *Enigma machine* provided a practical implementation of a polyalphabetic substitution cipher that were widely used during the period 1930-1950 for encryption. A rotor machine is an electro-mechanical device consisting of a series of rotors, each which has connections hard-wired to perform a simple monoalphabetic substitution. After each element is encoded the rotor advances one position and changes the substitution. Using just a single rotor would result in a key of length 26, so in order to confuse attempts at cryptanalysis the output from one rotor would be used as input to another which would rotate at a different rate. In practice, at least three rotors were used ensuring that there is no repetition in the pattern for $26^3 = 17576$ letters. However, there were successful cryptanalysis attacks during the period 1932-1945 on all but the most complex rotor machines, particularly by the Polish and British using algebraic techniques.

Transposition techniques rely on permuting the order of elements in the plaintext. For instance, a *rail fence cipher* writes plaintext elements row by row using say six columns then reads the elements column by column:

```
secure
andrel
iables
ystems
```
Ciphertext: `SAIYENASCBDTURLEREEMELSS`

This can be later decoded by writing the ciphertext column by column and reading off row by row. This algorithm could be improved by using a key that is a permutation of $(1, 2, 3, 4, 5, 6)$ to rearrange the columns before the ciphertext is read off:

```
secure          permute          csuree
andrel        (3,1,4,5,2,6)      darenl
iables           ⟹              bileas
ystems                           tyemss
```

Ciphertext: `CDBTSAIYURLEREEMENASELSS`

If the frequency of ciphertext elements were found to be similar to the expected frequencies in plaintext then a pure transposition cipher would be suspected by a cryptanalyst. If only a single transposition were used then it could be broken by guessing the number m of columns and testing their various $m!$ permutations. The encryption can however be made much harder to break by repeating the process multiple times using the same or several keys.

Exercise 1.1 (Steganography) *Rather than making a message unintelligible steganography instead tries to hide plaintext embedded inside an innocuous message. Prepare a program that can embed or extract a text message in an image by using the least-significant bit of the alpha (transparency) values for the image to store the plaintext bit by bit.*

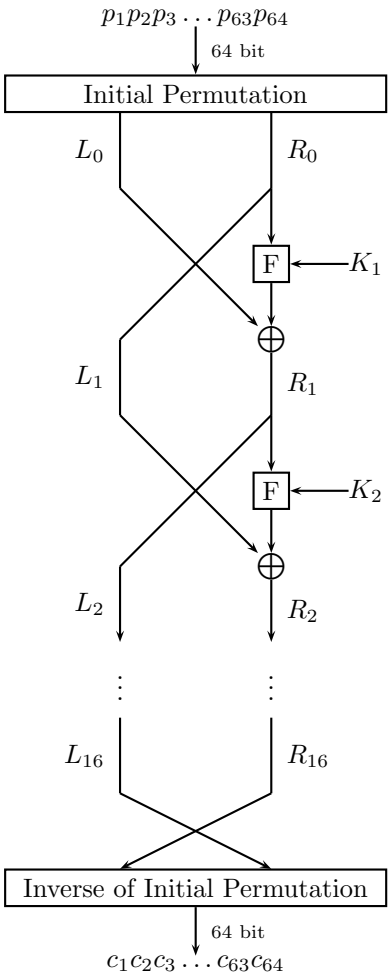
1.2 Data Encryption Standard

Contrary to some classical encryption techniques, usually no attempt is made to keep modern cryptography algorithms secret, most are widely known and so well tested for their security against known cryptanalysis attacks. If the encryption and decryption processes both use the same key then the encryption is known as *symmetric* or *single-key* encryption. If instead the encryption and decryption use different keys then the encryption is known as *asymmetric* or *public-key* encryption. Examples of symmetric ciphers include DES (discussed in this section), AES (Section 1.4), as well as triple DES and RC4 (Section 1.5), whereas RSA (Section 2.2) and ECC (Section 2.4) are both public-key ciphers. Encryption algorithms are also classified either as *block ciphers* if they process plaintext an entire block at a time, or else as *stream ciphers* if they can process plaintext one element at a time as the plaintext is read. For example, DES is a block cipher that typically processes 64-bit blocks of plaintext at a time, AES is another block cipher designed for 128-bit blocks, whereas RC4 is a stream cipher that processes the plaintext one byte at a time.

The *Data Encryption Standard* (DES) is the most widely-used symmetric cipher, although it is planned to eventually be replaced by more modern algorithms such as AES and triple-DES, which are presumed to be more resistant to cryptographic attacks. DES operates on 64-bit (8 byte) blocks of plaintext $p_1p_2p_3 \dots p_{63}p_{64}$ at a time to produce 64-bit $c_1c_2c_3 \dots c_{63}c_{64}$ ciphertext. The ciphertext is formed using a 64-bit key (of which only 56 bits are actually utilized). Firstly, the plaintext bits are permuted using a standard *initial permutation* giving bits $p_{58}p_{50}p_{42} \dots p_{15}p_7$, which is then split into two 32-bit halves L_0 and R_0. These two halves then pass through 16 rounds of substitutions and transpositions to progressively give L_1, R_1, then L_2, R_2, etc until L_{16}, R_{16} are obtained. At the start of each round the key is used to obtain a *subkey* K_i of 48 bits, by using various rotations and permutations of the bits of the key, and L_i, R_i are given by the recurrence:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \end{aligned}$$

where $\oplus$ denotes the exclusive-OR (XOR) bitwise operation (with $0 \oplus 0 = 0 = 1 \oplus 1$ and $0 \oplus 1 = 1 = 1 \oplus 0$) and F is a specific function for DES. Once L_{16}, R_{16} are obtained their order is swapped before they are combined back together to give 64 bits and the inverse of the initial permutation applied to finally give $c_1c_2c_3 \dots c_{63}c_{64}$.

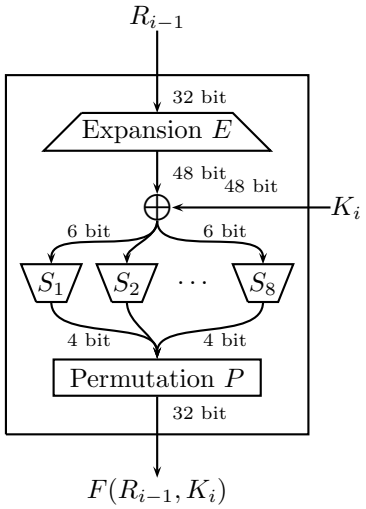


DES Encryption Algorithm

Since the initial permutation and its inverse are well-known they do not really contribute to the security of the algorithm, but the 16 rounds help diffuse any statistical structure that might be in the plaintext across the entire block and tries to confuse attempts at cryptanalysis.

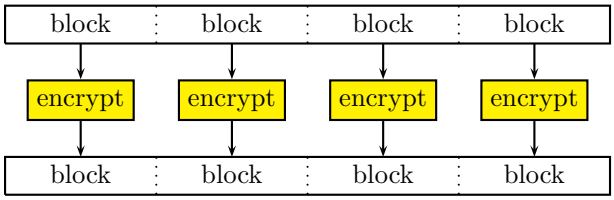
In each round the function F has as input a 32-bit $R_{i-1} = r_1r_2r_3 \ldots r_{31}r_{32}$ partially encrypted half of the block and a 48-bit subkey K_i. First it expands R_{i-1} by duplicating some bits to give a 48-bit $E(R_{i-1}) = r_{32}r_1r_2r_3 \ldots r_{31}r_{32}r_1$. This is then passed through a bitwise XOR with the subkey to give a 48-bit result. Next, the function F treats the 48 bits as eight groups of six bits, and for each group uses a special *S-box* to replace the six bits by four bits, effectively reducing the previous 48 bits to eight groups of four bits, or a 32-bit result. For example, if 000000 is in the first group it is replaced with 1110 from the S-box S_1, whereas if it is in the second group it is replaced with 1111 from the S-box S_2. Finally, these 32 bits are passed through a permutation P to give $F(R_{i-1}, K_i)$.

Although the eight S-boxes are well-known they have been a source of contention amongst cryptographers as their original design criteria were kept confidential. Hence suspicions arose as to whether they were designed with a weakness so that the US National Security Agency could decipher messages without knowledge of the key. Also of concern was the length of the key, only 56 bits, instead of the original IBM proposal of using a more-secure 128-bit key. Although this still allows $2^{56} \approx 7 \times 10^{16}$ possible keys, by 1998 DES-cracker devices were available that could break DES ciphertext via brute-force in less than three days, and by 2012 the keyspace could be exhausted in 26 hours with custom-built hardware.

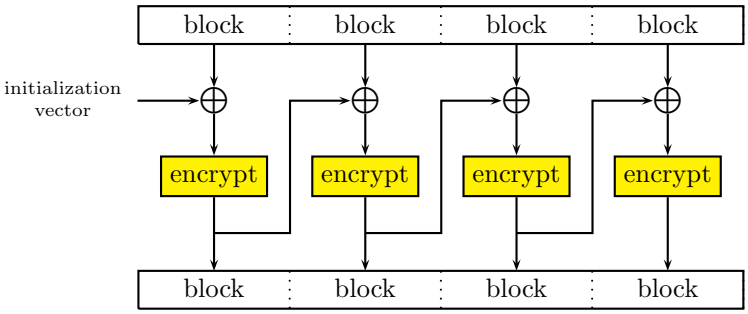


As DES is a symmetric cipher the same key is used to decrypt ciphertext as well as encrypt plaintext. Decryption is achieved by simply reversing the steps of the encryption, applying the subkeys in the reverse order and recalculating the $F(R_{i-1}, K_i)$. Hence if DES is implemented in hardware, the same hardware can be used for encryption as for decryption.

Any block cipher such as DES can be used with plaintext so long as its length is an exact multiple of the block size. The simplest way of encrypting multiple blocks of plaintext is known as *electronic code book* (ECB) mode, which simply encrypts each block separately (using the same key for the encryption of each block):



However, this has the disadvantage that identical plaintext blocks are encrypted as identical ciphertext blocks, which could reveal some patterns in the plaintext, particularly for longer plaintext. To remove this vulnerability, the *cipher block chaining* (CBC) mode takes a bitwise XOR of each plaintext block with the previous block's ciphertext before the plaintext block gets encrypted. An *initialization vector* of length one block is chosen and used for the bitwise XOR with the first block of plaintext. For added security the initialization vector should only be known to the sender and intended receiver (and might initially be sent to the receiver using ECB mode).



As the input for the encryption of each successive plaintext block is modified depending on the result of the previous encryption CBC ensures that identical plaintext blocks probably get encrypted to distinct ciphertext blocks.

The *cipher feedback* (CFB) and the *output feedback* (OFB) modes can each be used to convert a block cipher such as DES so that it can be used as a stream cipher that processes s bits at a time. Both these modes start with an initialization vector of length one block, encrypt it and select the s most-significant bits of the output. These s bits are passed through a bitwise XOR

with the first s bits of the plaintext stream to produce the first s-bit ciphertext. Then the bits in the initialization vector are shifted left by s bits. With CFB the s-bit ciphertext output is used as the new least significant bits of the modified vector, whereas with OFB the s most-significant bits of the encryption output are instead used (before they get passed to the XOR with the plaintext rather than after). Then the process is repeated using the modified vector to produce s-bit ciphertext for the next s bits of plaintext. Interestingly, decryption via CFB or OFB mode does not need to reverse the encryption algorithm, as it can just use the same process as for encryption to obtain the required s-bit outputs. One potential disadvantage of CFB is that any errors during transmission get propagated through the rest of the decryption process as each s-bit ciphertext received gets used in the following decryptions. Instead, with OFB a transmission error of one s-bit ciphertext affects only its corresponding s-bit plaintext, not any further plaintext, making it suitable for security over unreliable communication channels.

If the length of plaintext is not an exact multiple of the block size for a block cipher then *padding* needs to be added to fill out the remainder of the final block, which is removed when the ciphertext is later decrypted. Several padding schemes exist, such as simply appending zero bytes, or instead the more complicated and popular padding scheme *PKCS#7* which appends repeatedly with the number of bytes of padding and helps complicate an attack that might try to take advantage of the last block of ciphertext.

The *Java Cryptography Architecture* (JCA) is included as part of Java Standard Edition and include API in the packages `java.security`, `javax.crypto` and their subpackages that support encryption, decryption, and key generation. They allow for pluggable cryptographic security *providers* which provide implementations of various cryptographic algorithms, and include a default provider called `SunJCE`. The following steps are used for a symmetric block cipher:

Generate Key A key for symmetric encryption can be randomly generated by creating a suitable `KeyGenerator` for the encryption algorithm and using its `generateKey` method:

```
KeyGenerator kg = KeyGenerator.getInstance("DES");
SecretKey key = kg.generateKey();
```

Create Cipher For either encryption or decryption a `Cipher` object must first be created, specifying the name of the algorithm, and optionally its mode for handling multiple blocks and how it applies padding, such as:

```
Cipher cipher=Cipher.getInstance("DES/ECB/PKCS5Padding");
```

JCA supports modes `ECB` for electronic code book, `CBC` for cipher block chaining, `CFB` for cipher feedback, `OFB` for output feedback, as well as `PCBC` for propagating cipher block chaining, and includes `PKCS5Padding` and `NoPadding` for padding.

Initialize Cipher The cipher is then initialized either for encryption:

```
cipher.init(Cipher.ENCRYPT_MODE, key);
```

or else for decryption:

```
cipher.init(Cipher.DECRYPT_MODE, key);
```

Encrypt or Decrypt The `Cipher` method `doFinal` is used to encrypt or decrypt a `byte[]` array:

```
byte[] input = ...;
byte[] output = cipher.doFinal(input);
```

For example, the simple class `DESExample` demonstrates how DES encryption can be achieved using the JCA API.

DES encryption with Java Cryptography Architecture

```
/**
   A simple class that demonstrates how DES encryption is achieved using the Java
       Cryptography Architecture
   @author Andrew Ensor
*/
...
```

```
public class DESExample
{
   public static void main(String[] args)
   { try
      { // generate a secret key for DES
         KeyGenerator kg = KeyGenerator.getInstance("DES");
         SecretKey key = kg.generateKey();
         // create a cipher
         Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
         // obtain the plaintext
         Scanner keyboardInput = new Scanner(System.in);
         System.out.print("Please enter some plaintext:");
         byte[] plaintext = keyboardInput.nextLine().getBytes();
         // initialize cipher for encryption
         cipher.init(Cipher.ENCRYPT_MODE, key);
         // encrypt the plaintext
         byte[] ciphertext = cipher.doFinal(plaintext);
         System.out.print("Ciphertext is:");
         for (int i=0; i<ciphertext.length; i++)
         { int unsignedByte = ciphertext[i] & 0xFF;
            System.out.print(Integer.toHexString(unsignedByte)+" ");
         }
         System.out.println();
         // decrypt the ciphertext
         cipher.init(Cipher.DECRYPT_MODE, key);
         byte[] deciphertext = cipher.doFinal(ciphertext);
         System.out.println("Deciphered is:"+new String(deciphertext));
      }
      catch (NoSuchAlgorithmException e)
      { System.err.println("Encryption algorithm not available: "+e);
      }
      catch (NoSuchPaddingException e)
      { System.err.println("Padding scheme not available: "+e);
      }
      catch (InvalidKeyException e)
      { System.err.println("Invalid key: "+e);
      }
      catch (IllegalBlockSizeException e)
      { System.err.println("Cannot pad plaintext: "+e);
      }
      catch (BadPaddingException e)
      { System.err.println("Exception with padding: "+e);
      }
   }
}
```

Exercise 1.2 (Comparing ECB and CBC modes) *Use DES in ECB mode and then instead in CBC mode to encrypt the bytes in an image and display the plaintext image as well as both ciphertext images. Test with an image that has a uniform background.*

1.3 Finite Fields

The set $\{0, 1, 2, \ldots, n-1\}$ of the first n non-negative integers is usually denoted by $\mathbb{Z}_n$ and plays an interesting role in cryptography. Since adding together two integers each less than n might result in one that is greater than $n-1$ addition in $\mathbb{Z}_n$ is usually taken modulo n. For example $\mathbb{Z}_4 = \{0, 1, 2, 3\}$ has the following addition table:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$$\underbrace{a+b}_{\text{add in } \mathbb{Z}_n} = \underbrace{((\text{int})a + (\text{int})b)}_{\text{add in } \mathbb{Z}} \bmod n.$$

With this form of addition modulo n, $\mathbb{Z}_n$ gives an important example of what is known as an abelian group.

A *group* $(G, \bullet)$ is a set G together with a binary function $\bullet\colon G \times G \to G$ that obeys the following laws:

Associativity for all elements a, b, c in G one has $(a \bullet b) \bullet c = a \bullet (b \bullet c)$,

Identity there is an element e in G for which $a \bullet e = a = e \bullet a$ for every element a in G,

Inverse for every element a in G there is an element a' in G for which $a \bullet a' = e = a' \bullet a$.

If a group also satisfies:

Commutativity for all elements a, b in G one has $a \bullet b = b \bullet a$,

then the group is called *abelian* or *commutative*.

In cases where the group operation $\bullet$ is denoted by $+$, such as in the abelian group $\mathbb{Z}_n$ with addition modulo n, the identity e is usually denoted by 0 and the inverse a' of an element a by $-a$. Then a *subtraction* operation can be defined by $a - b = a + (-b)$.

Multiplication in $\mathbb{Z}_n$ can also be performed modulo n. For example $\mathbb{Z}_4$ has the following multiplication table:

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

$$\underbrace{a \cdot b}_{\text{mult in } \mathbb{Z}_n} = \underbrace{((\text{int})a \cdot (\text{int})b)}_{\text{mult in } \mathbb{Z}} \bmod n.$$

With both addition and multiplication taken modulo n, $\mathbb{Z}_n$ is an example of what is known as a commutative ring.

A *ring* $(R, +, \cdot)$ is an abelian group $(R, +)$ together with a second binary function $\cdot\colon R \times R \to R$ that obeys the following laws:

Associativity for all elements a, b, c in R one has $(a \cdot b) \cdot c = a \cdot (b \cdot c)$,

Distributivity for all elements a, b, c in R one has $a \cdot (b+c) = a \cdot b + a \cdot c$ and $(a+b) \cdot c = a \cdot c + b \cdot c$.

If a ring also satisfies:

Commutativity for all elements a, b in R one has $a \cdot b = b \cdot a$,

then the ring is said to be *commutative*.

The *greatest common divisor* of two positive integers m and n is the greatest positive number $\gcd(m, n)$ that divides both m and n. For example, the divisors of $m = 12$ are 1, 2, 3, 4, 6, 12, whereas the divisors of $n = 28$ are 1, 2, 4, 7, 14, 28. Hence, the common divisors are 1, 2, 4, so $\gcd(12, 28) = 4$. The Euclidean algorithm provides a simple procedure for determining the greatest common divisor using just the integer modulo operation.

Euclidean Algorithm for Positive Integers: Suppose m and n are positive integers. Then $\gcd(m, n) = \gcd(n \bmod m, m)$, and so $\gcd(m, n)$ can be calculated by taking repeated remainders $r_1 > r_2 > \cdots > r_n = \gcd(m, n)$ until one of the remainders r_n is a divisor of the previous r_{n-1}:

$$\begin{aligned}
r_1 &= n \bmod m && \text{where } r_1 < m \\
r_2 &= m \bmod r_1 && \text{where } r_2 < r_1 \\
r_3 &= r_1 \bmod r_2 && \text{where } r_3 < r_2 \\
r_4 &= r_2 \bmod r_3 && \text{where } r_4 < r_3 \\
&\vdots && \vdots \\
r_n &= r_{n-2} \bmod r_{n-1} && \text{where } r_n < r_{n-1} \\
0 &= r_{n-1} \bmod r_n.
\end{aligned}$$

Furthermore, there are integers s and t for which $\gcd(m, n) = s \cdot m + t \cdot n$.

For example, $\gcd(945, 2415)$ can be found as follows:

$$\begin{aligned} 2415 &= 2 \cdot 945 + 525 && \text{so } r_1 = 525 \\ 945 &= 1 \cdot 525 + 420 && \text{so } r_2 = 420 \\ 525 &= 1 \cdot 420 + 105 && \text{so } r_3 = 105 \\ 420 &= 4 \cdot 105 + 0 && \text{so } r_4 = 0. \end{aligned}$$

Hence $\gcd(945, 2415) = 105$. The values of s and t can then be found from these equations using them in reverse order:

$$\begin{aligned} 105 &= 525 + (-1) \cdot 420 \\ &= 525 + (-1) \cdot (945 + (-1) \cdot 525) \\ &= 2 \cdot 525 + (-1) \cdot 945 \\ &= 2 \cdot (2415 + (-2) \cdot 945) + (-1) \cdot 945 \\ &= (-5) \cdot 945 + 2 \cdot 2415. \end{aligned}$$

Division in a ring can be more problematic than addition, subtraction, and multiplication. For instance since $2 \cdot 2 = 0$ in $\mathbb{Z}_4$, it is not possible to have a meaningful way to divide elements by 2 (otherwise one could divide both sides of the equation $2 \cdot 2 = 0$ by 2 to obtain that $2 = 0$ which is a contradiction). This problem arises whenever there are elements $a \neq 0$ and $b \neq 0$ in the ring for which $a \cdot b = 0$. In the case of the ring $(\mathbb{Z}_n, +, \cdot)$ this problem can occur if and only if n is not a prime number.

If a commutative ring also satisfies:

Identity there is an element 1 in R for which $a \cdot 1 = a = 1 \cdot a$ for every element a in R,

Inverse for every element $a \neq 0$ in R there is an element a^{-1} for which $a \cdot a^{-1} = 1 = a^{-1} \cdot a$.

then the ring is called a *field*. Essentially, a field is a ring in which division makes sense and can be defined by $a/b = a \cdot b^{-1}$ when $b \neq 0$.

From the previous comments one can conclude that $\mathbb{Z}_n$ with addition and multiplication modulo n is a field if and only if n is a prime number. For example, since 5 is a prime $\mathbb{Z}_5$ is a field. This can be seen from its multiplication table as for $a \neq 0$ and $b \neq 0$ one has $a \cdot b \neq 0$ in $\mathbb{Z}_5$:

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

a	$-a$
0	0
1	4
2	3
3	2
4	1

$\cdot$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

a	a^{-1}
0	none
1	1
2	3
3	2
4	4

The inverse a^{-1} can be found from the row of the multiplication table for the element a by locating the column that has the identity element 1. However, it can also be calculated directly in $\mathbb{Z}_p$ without first finding the multiplication table by using the Euclidean algorithm. Since $\gcd(a, p) = 1$, and the values s and t satisfy $\gcd(a, p) = s \cdot a + t \cdot p$, it follows that $s \bmod p = a^{-1}$. For example, 12^{-1} can be found in $\mathbb{Z}_{17}$ by:

$$\begin{aligned} 17 &= 1 \cdot 12 + 5 && \text{so } r_1 = 5 \\ 12 &= 2 \cdot 5 + 2 && \text{so } r_2 = 2 \\ 5 &= 2 \cdot 2 + 1 && \text{so } r_3 = 1 \\ 2 &= 2 \cdot 1 + 0 && \text{so } r_4 = 0. \end{aligned}$$

Hence $\gcd(12, 17) = 1$ as expected, and s and t are found by:

$$\begin{aligned} 1 &= 5 + (-2) \cdot 2 \\ &= 5 + (-2) \cdot (12 + (-2) \cdot 5) \\ &= 5 \cdot 5 + (-2) \cdot 12 \\ &= 5 \cdot (17 + (-1) \cdot 12) + (-2) \cdot 12 \\ &= (-7) \cdot 12 + 5 \cdot 17. \end{aligned}$$

Hence $12^{-1} = (-7) \bmod 17 = 10$ in $\mathbb{Z}_{17}$ (and indeed $12 \cdot 10 \bmod 17 = 1$).

Fields with a finite set of elements are important in cryptography, particularly finite fields where there are 2^m elements rather than p elements for some prime p. Fortunately, for every prime p and positive integer m there is a field denoted by $GF(p^m)$ with p^m elements, and called the *Galios field* of order p^m.

One of the most intuitive ways of understanding the addition $+$ and multiplication $\cdot$ operations in the field $GF(p^m)$ is to consider each of the elements $a = (a_0, a_1, a_2, \ldots, a_{m-1})$ in $GF(p^m)$ as a polynomial $f(x) = a_{m-1}x^{m-1} + \cdots + a_2x^2 + a_1x + a_0$, where a_0, a_1, $\ldots$, a_{m-1} are elements in $\mathbb{Z}_p$. Then addition in $GF(p^m)$ is just the addition of two polynomials whose coefficients are added together modulo p. For example, in the field $GF(5^4)$ the polynomials $f(x) = 3x^3 + 1x^2 + 0x + 4$ and $g(x) = 4x^3 + 2x^2 + 3x + 2$ can be considered as elements whose sum is $f(x) + g(x) = 2x^3 + 3x^2 + 3x + 1$.

Multiplication in $GF(p^m)$ is slightly more complicated as the product of two polynomials in $GF(p^m)$ might produce a polynomial of degree greater than $m-1$, but simply discarding higher terms would not result in a field. For example $\big(x^3 + 0x^2 + 0x + 0\big) \cdot (x + 0) = x^4$ which should not be considered the zero element $0x^3 + 0x^2 + 0x + 0$ in $GF(5^4)$. What is needed when a product results in a polynomial of degree greater than $m - 1$ is to take the remainder when the product is divided by a polynomial that acts much like a prime number. A polynomial of degree m is called *reducible* in $GF(p^m)$ if it factorizes as a product $f(x) \cdot g(x)$ of two polynomials that are in $GF(p^m)$, otherwise it is called *irreducible* in $GF(p^m)$. For example, $x^4 + x^2 + 1$ is reducible in $GF(5^4)$ as $x^4 + x^2 + 1 = \big(x^2 + x + 1\big) \cdot \big(x^2 + 4x + 1\big)$. Similarly, x^4 is reducible as $x^4 = x^3 \cdot x$, but $x^4 + 2$ can be shown to be irreducible in $GF(5^4)$. Once an irreducible polynomial $q(x)$ has been found multiplication in $GF(p^m)$ can be defined by:

$$\underbrace{f(x) \cdot g(x)}_{\text{mult in } GF(p^m)} = \underbrace{f(x) \cdot g(x)}_{\text{mult as polys}} \bmod q(x).$$

Choosing a different irreducible polynomial results in a slightly different multiplication operation, but surprisingly gives essentially the same field but with the elements permuted. In fact it can be shown that the fields $GF(p^m)$ are the only finite fields that are possible (mathematically, any finite field must be *isomorphic* to $GF(p^m)$ for some prime p and $m \geq 1$, meaning that they are essentially the same apart from what the elements are called).

Note that the field $\mathbb{Z}_p$ is just a special case of $GF(p^m)$ where $m = 1$. Also, inverses of elements in $GF(p^m)$ can be calculated directly using a polynomial equivalent of the Euclidean Algorithm.

The field $GF(2^8)$ is often chosen for cryptography applications as it has $2^8 = 256$ elements, one element for each possible byte value. The elements can be considered as polynomials $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ where a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7 are elements of $\mathbb{Z}_2$. Addition in $\mathbb{Z}_2$ is simply XOR so addition of two polynomials in $GF(2^8)$ is simply just a bitwise XOR operation of the corresponding coefficients. For example, adding together $x^7 + x^4 + x$ and $x^5 + x^4 + x + 1$ results in $x^7 + x^5 + 1$.

It can be shown that there are 30 possible irreducible polynomials of degree 8 in $GF(2^8)$, such as the polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$ which is the polynomial chosen for the AES cipher (discussed in Section 1.4). Using this irreducible polynomial, the product of $x^7 + x^4 + x$ and $x^5 + x^4 + x + 1$ in $GF(2^8)$ is given by:

$$\begin{aligned}
\overbrace{\big(x^7 + x^4 + x\big) \cdot \big(x^5 + x^4 + x + 1\big)}^{\text{mult in } GF(2^8)} &= \big(x^{12} + x^{11} + x^8 + x^7 \\
&\quad + x^9 + x^8 + x^5 + x^4 \\
&\quad + x^6 + x^5 + x^2 + x\big) \bmod q(x) \\
&= \big(x^{12} + x^{11} + x^9 + x^7 \\
&\quad + x^6 + x^4 + x^2 + x\big) \bmod q(x) \\
&= x^7 + x^4 + x + 1,
\end{aligned}$$

where the remainder of dividing $x^{12}+x^{11}+x^9+x^7+x^6+x^4+x^2+x$ by $q(x) = x^8+x^4+x^3+x+1$ can be found using long division.

Exercise 1.3 (The Finite Field $GF(2^2)$) *Determine the addition, negative, multiplication, and inverse tables for the finite field $GF(2^2)$ which consists of the four elements 0, 1, x, $x+1$, using the irreducible polynomial $q(x) = x^2 + x + 1$ for the multiplication operation.*

1.4 Advanced Encryption Standard

As computing power increased the DES cipher with a key length of only 56 bits became more and more vulnerable to a brute-force attack, so a competition was held where new algorithms were suggested for its eventual replacement. During the competition the cipher algorithms could be openly scrutinized for susceptibility to cryptanalysis attacks and their speed and ease of implementation compared. In 2000 a symmetric block cipher algorithm known as *Rijndael* was selected as the new *Advanced Encryption Standard* (AES) cipher. This was considered a landmark for security as it demonstrated that open scrutiny of an encryption algorithm could result in better security than a secretly developed algorithm.

AES operates on blocks of length 128 bits (16 bytes), double that of DES, although the Rijndael algorithm also allows for block sizes of 192 or 256 bits. Keys can be of length 128, 192, or 256 bits. It treats a 16-byte block of plaintext as a 4×4 matrix of bytes arranged column by column, so that the bytes `73 65 63 75 72 65 20 61 6E 64 20 72 65 6C 69 61` (for the ASCII string `"secure and relia"`) are represented by the matrix:

$$\begin{pmatrix} \texttt{73} & \texttt{72} & \texttt{6E} & \texttt{65} \\ \texttt{65} & \texttt{65} & \texttt{64} & \texttt{6C} \\ \texttt{63} & \texttt{20} & \texttt{20} & \texttt{69} \\ \texttt{75} & \texttt{61} & \texttt{72} & \texttt{61} \end{pmatrix}.$$

Firstly, the matrix is passed through a bitwise XOR with the 128-bit key K (also arranged as a 4×4 byte matrix). Then the matrix passes through 10 rounds, each consisting of three operations (except the final tenth round which has just two operations), each followed by a bitwise XOR using a key K_i that is obtained from the original key K and which gets modified with each round.

The first operation in each round performs a simple substitution of each byte in the matrix using an S-box:

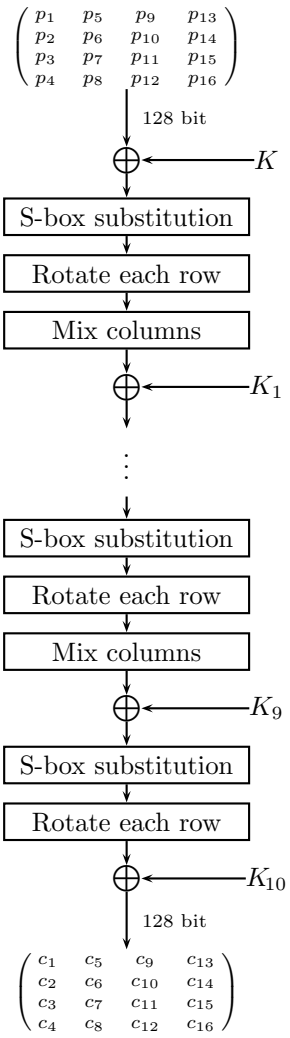
AES S-box substitutions

Byte	*Substitution*
00	63
01	7C
02	77
03	7B
04	F2
⋮	⋮
FF	16

Unlike the obscure and secretive approach of DES S-boxes, it is well-known how the AES S-box was designed. Firstly, each byte $a_7a_6a_5a_4a_3a_2a_1a_0$ is considered as a polynomial element $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ in the finite field $GF(2^8)$. For all but the zero byte, its inverse in $GF(2^8)$ is found, giving another byte $b_7b_6b_5b_4b_3b_2b_1b_0$ which is then fed into the following equation using $\mathbb{Z}_2$ arithmetic to give the S-box substitution byte:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The AES S-box gives an invertible permutation of the bytes designed to be resistant to currently known cryptanalysis attacks and optimized to ensure a low correlation between each byte and its



AES Encryption Algorithm (above)

substituted byte. For performance, the S-box and its inverse (which is used for AES decryption) are usually stored in memory rather than being recalculated each time for the algorithm.

The second operation in each round rotates the bytes row by row in the matrix. The first row is left unchanged, whereas the second row is rotated one entry to the left, the third row by two entries to the left, and the fourth row by three entries to the left, resulting in a transposition of the bytes in the block:

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \overset{\text{Rotate}}{\Longrightarrow} \begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_6 & m_{10} & m_{14} & m_2 \\ m_{11} & m_{15} & m_3 & m_7 \\ m_{16} & m_4 & m_8 & m_{12} \end{pmatrix}.$$

The third operation that is used in the first nine rounds multiplies the matrix on the left by the invertible matrix:

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

where again each byte $a_7a_6a_5a_4a_3a_2a_1a_0$ is considered as a polynomial element $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ in the finite field $GF(2^8)$ for the purpose of addition and matrix multiplication (so 01 is the constant polynomial 1, 02 is the polynomial x, and 03 is the polynomial $x + 1$). These particular entries in the matrix were chosen since multiplication in $GF(2^8)$ by the element 02 or 03 can be implemented using at most a bit-shift and an XOR (the entries in its inverse require more sophisticated implementations, but in practice encryption is often performed more frequently than decryption, such as when CFB or OFB cipher modes are used). Also, the entries ensure that after a few rounds of row rotations and mix columns all the output bits depend on all the input bits of the plaintext block.

If the key used for a round is the matrix:

$$\begin{pmatrix} k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \\ k_4 & k_8 & k_{12} & k_{16} \end{pmatrix}$$

(starting with the original key K) then the key used for the next round is given by:

$$\begin{pmatrix} k'_1 & k'_5 & k'_9 & k'_{13} \\ k'_2 & k'_6 & k'_{10} & k'_{14} \\ k'_3 & k'_7 & k'_{11} & k'_{15} \\ k'_4 & k'_8 & k'_{12} & k'_{16} \end{pmatrix}$$

where for $i \geq 5$ the entries are given by $k'_i = k'_{i-4} \oplus k_i$ (each entry is the XOR of the entry to its left with the corresponding entry in the previous key). For the first column of the new key a more complicated procedure is used to ensure the key changes round by round:

$$\begin{aligned} k'_1 &= \text{Sub}(k_{14}) \oplus R \oplus k_1 \\ k'_2 &= \text{Sub}(k_{15}) \oplus R \oplus k_2 \\ k'_3 &= \text{Sub}(k_{16}) \oplus R \oplus k_3 \\ k'_4 &= \text{Sub}(k_{13}) \oplus R \oplus k_4 \end{aligned}$$

where Sub denotes the S-box substitution and R is a byte that depends on the round.

R byte values for calculating new key each round

Round	1	2	3	4	5	6	7	8	9	10
R byte	01	02	04	08	10	20	40	80	1B	36

Although the decryption algorithm is just the reverse of the encryption algorithm, it must undo the operations in the reverse order to the encryption algorithm, and so has a separate implementation from encryption. However, one feature of AES is that by slightly changing how the keys are calculated for each round during decryption the steps can be reversed using the same order in which they were used for encryption, enabling encryption and decryption to share much of their implementation details (this is actually made possible by having the last round of encryption not apply the mix columns operation).

The default JCA cryptography provider SunJCE supports AES, and the example from Section 1.2 can be modified to use AES simply by replacing `DES` by `AES` in the `getInstance` method of `KeyGenerator` and of `Cipher`. As AES supports different key lengths some cryptographic security providers might support 192-bit or 256-bit keys as well as the default 128-bit keys. Different length keys can be produced by `KeyGenerator` by including the statement:

```
kg.init(128);
```

before the key is generated. The class `AESExample` uses AES encryption but in CBC mode rather than the default ECB mode, and so requires an initialization vector. An initialization vector is created using an `IvParameterSpec` and specifying a `byte[]` array:

```
byte[] ivBytes = ...; // array of 16 bytes
IvParameterSpec initVector = new IvParameterSpec(ivBytes);
```

The `IvParameterSpec` can then be included as a parameter to the `Cipher` method `init` whenever the cipher is initialized for encryption or for decryption:

```
cipher.init(Cipher.ENCRYPT_MODE, key, initVector);
```

If an initialization vector is not specified to the `Cipher` method `init` and it is being initialized for encryption then it generates a suitable random vector itself, whose `byte[]` array can then be obtained from the `Cipher` method `getIV`. If however it is being initialized for decryption then it will throw an `InvalidKeyException` if an initialization vector is required but not specified.

AES encryption with CBC mode

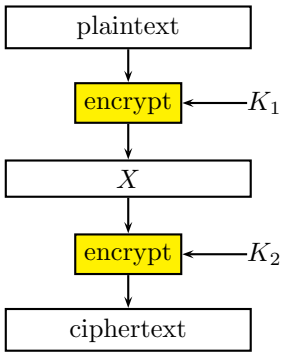
```
/**
   A simple class that demonstrates how AES encryption with CBC mode is achieved using
       the Java Cryptography Architecture
   @author Andrew Ensor
*/
...
public class AESExample
{
   public static void main(String[] args)
   { try
      { // generate a secret key for AES
         KeyGenerator kg = KeyGenerator.getInstance("AES");
         kg.init(128); // 128-bit key used for AES
         SecretKey key = kg.generateKey();
         // create a cipher
         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
         // obtain the plaintext
         ...
         // create an initialization vector (required for CBC)
         byte[] ivBytes = {51, 50, 7, -19, 120, 111, -110, 52, 9, -21,
            -6, -15, -95, 117, 36, -89}; // random array of 16 bytes
         IvParameterSpec initVector = new IvParameterSpec(ivBytes);
         // initialize cipher for encryption
         cipher.init(Cipher.ENCRYPT_MODE, key, initVector);
         // encrypt the plaintext
         byte[] ciphertext = cipher.doFinal(plaintext);
         ...
         // decrypt the ciphertext
         cipher.init(Cipher.DECRYPT_MODE, key, initVector);
         byte[] deciphertext = cipher.doFinal(ciphertext);
```

```
        System.out.println("Deciphered is:"+new String(deciphertext));
      }
      catch (NoSuchAlgorithmException e)
      { System.err.println("Encryption algorithm not available: "+e);
      }
      catch (NoSuchPaddingException e)
      { System.err.println("Padding scheme not available: "+e);
      }
      catch (InvalidKeyException e)
      { System.err.println("Invalid key: "+e);
      }
      catch (InvalidAlgorithmParameterException e)
      { System.err.println("Invalid algorithm parameter: "+e);
      }
      catch (IllegalBlockSizeException e)
      { System.err.println("Cannot pad plaintext: "+e);
      }
      catch (BadPaddingException e)
      { System.err.println("Exception with padding: "+e);
      }
   }
}
```

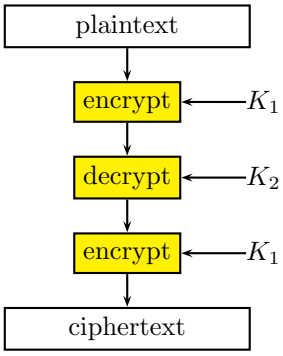
Exercise 1.4 (Performance of DES and AES) *Prepare a program that can be used to compare the performance of the DES cipher with the AES cipher for different length plaintext and plot a graph of time versus length.*

1.5 Other Symmetric Ciphers

The DES cipher is gradually being phased out and replaced by more secure ciphers such as AES with a longer key and so less susceptible to brute-force attacks. However, AES is a newer and completely different algorithm which has not yet had the same amount of cryptanalysis testing as has DES. Furthermore, there has been a significant software and hardware investment in the DES algorithm. Until AES is fully accepted as the defacto symmetric block cipher a popular alternative has been to apply the DES encryption several times using different keys.



One attempt at improving DES might be to apply it twice, once to the plaintext with one 56-bit key K_1 and then to that ciphertext X with another 56-bit key K_2, essentially resulting in a 112-bit key. This way a basic brute-force attack would take 2^{56} times longer than a brute-force attack on single DES, making it infeasible with current technology. However, simply applying a block cipher such as DES twice can be susceptible to the *meet-in-the-middle* attack if some blocks of plaintext and its corresponding ciphertext are known to a cryptanalyst. This attack works on double DES by taking a block of plaintext and encrypting it with all 2^{56} possible values for K_1, storing the resulting possible X blocks in a suitable data structure along with the key used. It then takes the corresponding ciphertext block and decrypts it with all 2^{56} possible values for K_2 and checks if each resulting decrypted block is already in the data structure. Whenever the same value of X is obtained the keys used for encryption and for decryption give potential values for K_1 and K_2. Since this could happen various times by chance the potential values for K_1 and K_2 are then checked with another known plaintext and ciphertext.



Because of the meet-in-the-middle attack three stages are instead used with DES. *Triple DES* with two keys K_1 and K_2 (112-bit key) is widely used, where K_1 is used in the first stage to encrypt the plaintext, which then gets decrypted but by using K_2 (which effectively scrambles it more), and then it is encrypted using K_1 again in the third stage. Decryption is chosen rather than encryption in the second stage so that single DES encryption can be obtained (if required) by setting $K_1 = K_2$. Since there is a potential attack (albeit impractical as it requires 2^{56} plaintext and corresponding ciphertext blocks to be available to a cryptanalyst), triple DES is now often used with three keys, where the third key is used in the last stage rather than again using K_1. However, the required 168-bit key is quite a bit longer than 128-bit keys used by other algorithms presumed to be at least as secure, and DES is itself considered quite slow so applying it three times over for each block makes the encryption process even three times slower.

Another popular symmetric block cipher is *Blowfish*, designed in 1993 as an open and general-purpose replacement for DES. It uses block sizes of 64 bits and a key length between 32 and 448 bits (with default 128-bit key). Its algorithm has a similar structure to DES, also using 16 rounds with two 32-bit halves, but it uses pre-computed key-dependent S-boxes. This makes the algorithm quite fast when the same key can be reused multiple times. Although there is no publicly-known successful attack on Blowfish, block ciphers that use larger blocks than the 64-bit blocks used by DES, triple DES, and Blowfish are generally now preferred. As a consequence, Blowfish evolved to become *Twofish*, which uses 128-bit blocks (the same as the AES Rijndael algorithm), and key length between 128 and 256 bits. On most platforms Twofish is slightly slower than the Rijndael algorithm for 128-bit keys but is faster for 256-bit keys.

JCA supports triple DES with keys of length 112 (two key triple DES) and 168 (three key triple DES) using the algorithm name `DESede`. It also supports Blowfish with the algorithm name `Blowfish`.

The most popular stream cipher in use is the *Rivest Cipher 4* (RC4) cipher. It has a remarkably fast and simple algorithm that can use a key length anywhere between 8 and 2048 bits (1 to 256 bytes), although at the very minimum 40 bits (5 bytes) should be used to produce a *key stream* of bytes. To encrypt the plaintext bytes, they are passed through a bitwise XOR with the key stream bytes. The encrypted bytes can then be decrypted by passing them again through a bitwise XOR with the same key stream to recover the plaintext bytes. The security of RC4 relies on the key stream bytes being unpredictable without knowledge of the key that produced the stream. RC4 starts by initializing a state array S, which is a `byte[]` array of length 256 that holds some unpredictable permutation of the bytes 0, 1, 2, ..., 255. The initial permutation of the bytes is determined by a key K. The array S is used to determine which byte is next used in the key stream. Instead of simply iterating through the bytes in S using a single index variable a (which would result in the key stream repeating itself after 256 bytes), another variable b is also used to help determine which byte is next used and to further permute the array S with each byte generated.

Although RC4 is not considered to provide the same level of security as ciphers such as AES, and the key stream it produces has been shown to be biased in terms of certain sequences, provided a reasonable length key (such as 128 bits) is used it is considered reasonably secure for some practical uses. Due to its encryption/decryption speed and its comparative simplicity, the RC4 stream cipher was adopted for use in the Secure Sockets Layer/Transport Layer Security (SSL/TLS) standards, in the Wired Equivalent Privacy (WEP) protocol for 802.11 wireless network security, and in the WiFi Protected Access (WPA) protocol. However, its implementation in the WEP protocol used a flawed method to generate keys, enabling the protocol to be compromised in 2001 by an attack due to Fluhrer, Mantin, and Shamir.

When an encryption technique, such as RC4, is performed by passing the plaintext through an XOR with a key stream it is important that different keys be used to encrypt each plaintext. Otherwise a cryptanalyst could perform a bitwise XOR of two encrypted messages to obtain an XOR of the two original plaintext, and so obtain information where the plaintext differ from

RC4-INITIALIZATION(K)

```
1  ▷ initialize the state array S to contain the bytes 0, 1, 2, ..., 255
2  for i ← 0 to 255 do
3          S[i] ← i
4  ▷ use key K to permute the bytes in S
5  m ← length[K]
6  j ← 0
7  for i ← 0 to 255 do
8          j ← (j + S[i] + K[i mod m]) mod 256
9          swap S[i] and S[j]
10 ▷ initialize the index pointers a and b
11 a ← 0
12 b ← 0
```

RC4-GETNEXTBYTE()

```
1  ▷ update the index pointers a and b
2  a ← (a + 1) mod 256
3  b ← (b + S[a]) mod 256
4  swap S[a] and S[b]                    ▷ further permute the bytes in S
5  t ← (S[a] + S[b]) mod 256                    ▷ next index of S to use
6  return S[t]                           ▷ next byte to use in key stream
```

each other.

For convenience, JCA offers the classes `CipherInputStream` and `CipherOutputStream` that act as filtering streams, providing encryption of a data stream. Both filter streams accept an initialized `Cipher` object passed to their constructor. Interestingly, the `Cipher` can itself be a block cipher such as DES or AES. If a block cipher is used with `CipherOutputStream` one should ensure that the `CipherOutputStream` method `close` is called (rather than just `flush`) when the bytes have all been written to the output stream, since if the total number of bytes written is not an exact multiple of the block size the `flush` method does not pad the last block so the block is still not written out.

The class `RC4Example` uses JCA cipher streams with an RC4 stream cipher given the name `ARCFOUR` (for Alleged-RC4 as RSA never officially acknowledged that this is the true RC4 algorithm). This example also demonstrates how a desired key can be created using the `SecretKeySpec` class rather than randomly via a `KeyGenerator`. Note however that secret information such as the key should never in practice be stored in a `String` (as any `String` is immutable its contents cannot be changed after use).

RC4 ENCRYPTION

```
/**
   A simple class that demonstrates how RC4 encryption is achieved using the Java
       Cryptography Architecture
   @author Andrew Ensor
*/
...
public class RC4Example
{
   public static void main(String[] args)
   { try
      { // create a secret key for RC4 (using bytes of "Secret")
        // Note in practice information related to the key such as in
        // the next line should NEVER be stored in a String
        SecretKey key = new SecretKeySpec("Secret".getBytes(),
           "ARCFOUR");
        // create a cipher
        Cipher cipher = Cipher.getInstance("ARCFOUR");
        // obtain the plaintext
        Scanner keyboardInput = new Scanner(System.in);
```

```
        System.out.print("Please enter some plaintext:");
        byte[] plaintext = keyboardInput.nextLine().getBytes();
        // initialize cipher for encryption
        cipher.init(Cipher.ENCRYPT_MODE, key);
        // encrypt the plaintext and store in a List
        CipherInputStream cis = new CipherInputStream
           (new ByteArrayInputStream(plaintext), cipher);
        List<Integer> ciphertext = new ArrayList<Integer>();
        System.out.print("Ciphertext is:");
        int unsignedByte = cis.read();
        while (unsignedByte != -1)
        { ciphertext.add(unsignedByte);
           System.out.print(Integer.toHexString(unsignedByte)+" ");
           unsignedByte = cis.read();
        }
        System.out.println();
        cis.close();
        // initialize cipher for decryption
        cipher.init(Cipher.DECRYPT_MODE, key);
        // decrypt the ciphertext and store in ByteArrayOutputStream
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        CipherOutputStream cos = new CipherOutputStream(baos, cipher);
        for (int data : ciphertext)
           cos.write(data);
        cos.close(); // don't just flush if using a block cipher
        byte[] deciphertext = baos.toByteArray();
        System.out.println("Deciphered is:"+new String(deciphertext));
     }
     ...
  }
}
```

The .NET platform provides implementations of some common cipher algorithms in the `System.Security.Cryptography` namespace, including DES with the class `DESCryptoServiceProvider`, AES with the class `RijndaelManaged` (a managed wrapper on top of the Windows CAPI service), and triple DES with the class `TripleDESCryptoServiceProvider`. There is no block mode of encryption or decryption in C#, all encryption is performed using streams with a `CryptoStream` filtering stream, even when using a block cipher. When a `CryptoStream` is created it is provided with an `ICryptoTransform` obtained from the cipher that can either encrypt or else decrypt, and the `CryptoStream` is created to either read from an input stream or else write to an output stream:

```
Stream outputStream = ...;
CryptoStream cs = new CryptoStream(outputStream,
   cipher.CreateEncryptor(), CryptoStreamMode.Write);
```

The C# class `NETExample` demonstrates how AES encryption can be performed in C#. This example creates a `RijndaelManaged` (AES) cipher which itself generates a random 128-bit key and initialization vector. The key and initialization vector are held by the `Key` and `IV` properties of the cipher, and get stored in variables in the example so that they can be specified when the ciphertext is later decrypted.

AES encryption in C#

```
/*
   A C# class that demonstrates how AES encryption is achieved in .NET
*/
using System;
using System.Collections;
using System.IO;
using System.Security.Cryptography;
using System.Text;
```

```
public class NETExample
{
   public static void Main(string[] args)
   { // create UTF8 encoder for converting user string to/from byte[]
      UTF8Encoding encoder = new UTF8Encoding();
      try
      { // create a cipher and have it generate a secret key
         RijndaelManaged cipher = new RijndaelManaged();
         byte[] key = cipher.Key;
         byte[] initVector = cipher.IV;
         // obtain the plaintext
         Console.Write("Please enter some plaintext:");
         byte[] plaintext = encoder.GetBytes(Console.ReadLine());
         // encrypt the plaintext and store in a List
         CryptoStream cis = new CryptoStream(
            new MemoryStream(plaintext), cipher.CreateEncryptor(),
            CryptoStreamMode.Read);
         IList ciphertext = new ArrayList();
         Console.WriteLine("Ciphertext is:");
         int unsignedByte = cis.ReadByte();
         while (unsignedByte != -1)
         { ciphertext.Add(unsignedByte);
            Console.Write("{0:X} ",unsignedByte);
            unsignedByte = cis.ReadByte();
         }
         Console.WriteLine();
         cis.Close();
         // decrypt the ciphertext and store in MemoryStream
         MemoryStream ms = new MemoryStream(100);
         CryptoStream cos = new CryptoStream(ms,
            cipher.CreateDecryptor(key, initVector),
            CryptoStreamMode.Write);
         foreach (int data in ciphertext)
            cos.WriteByte((byte)data);
         cos.Close();
         byte[] deciphertext = ms.GetBuffer();
         Console.WriteLine("Deciphered is:"
            + encoder.GetString(plaintext));
      }
      catch (CryptographicException e)
      { Console.WriteLine("Cryptographic exception: "+e);
      }
   }
}
```

Exercise 1.5 (Implementation of RC4) *Prepare a program that provides an implementation of the RC4 stream cipher (note that the* `%` *modulo operation only performs as expected for positive values, so before performing* `x % 256` *repeatedly add* 256 *to* `x` *to ensure that it is not negative).*

1.6 Secret Key Distribution

For a distributed computer system there are two common alternative approaches for encrypting messages to ensure their confidentiality in the network:

link encryption where every vulnerable communication link has an encryption device at each end, encrypting each message before it gets transmitted along the link and decrypting it at the other end,

end-to-end encryption where the sender of the message encrypts the message and the destination decrypts it when it is received.

Link encryption might result in a message being encrypted and decrypted various times if it

passes along several vulnerable links, however the encryption process is often performed by hardware devices which can help reduce transmission delays. These devices might also perform *traffic padding*, transmitting random messages when the link is not being used to confuse attempts that try to deduce when the link is transmitting important messages. Besides the transmission delays another drawback to using link encryption is that it requires a large number of encryption devices covering every possible message path, which might be infeasible in a large network (such as the Internet). Also, if one of the devices could be compromised the plaintext messages could be obtained. End-to-end encryption would typically be performed either at the network layer (encrypting the contents of IP packets) or else at the application layer (by application software). This approach ensures that user messages are secure even across insecure links, but since the messages must be forwarded by routers around the network the sender and recipient of each message remain exposed.

In practice a combination of both link encryption and end-to-end encryption is used to achieve greater security. If a symmetric cipher is employed in either approach then the encrypting end and the decrypting end must share a secret key. Each encrypting/decrypting pair should use its own key, so that if an attack successfully obtained one of the keys only one part of the network would be compromised. However, this creates the problem of how to distribute secret keys to each pair, which in a distributed system with n processes could require a separate key for each of the $\frac{n(n-1)}{2}$ possible pairs of processes that might communicate with each other.

Confidential information about a key must not be sent as unencrypted plaintext across a network so keys must be physically delivered to the relevant processes in some secure way. For link encryption this is a reasonable requirement as each encryption device only needs keys to perform encryption with its neighbouring devices, which could be provided when the communication link is installed. But for end-to-end encryption the physical distribution of all required keys become impractical given the possibly large number of processes with which a process might communicate, adding a new process to the system would require physical delivery of a new key to each of the existing processes, hindering the system's scalability.

Instead of physically distributing each key required by a process, a large distributed system can have one or more distinguished processes known as a *key distribution centre*. Each process is physically delivered a single secret *master key* which it can use to encrypt/decrypt messages to/from the key distribution centre, which could be provided when the process is first added to the distributed system. Thus the key distribution centre is able to communicate securely with each process in the system using the correct master key for symmetric encryption with that process. More than one key distribution centre might be used to improve the reliability of the system in case one centre fails or is compromised.

Rather than performing all secure communication via the key distribution centre, which would quickly become a bottleneck in the system, when a process wants to initiate secure communication with another process (receiver) it uses the following steps:

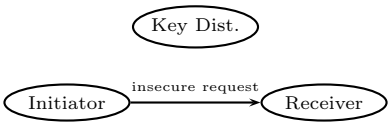
1. The initiator sends a request encrypted using its master key to the key distribution centre asking for a *session key* for its communication with the receiver, including a chosen *nonce value* that is not easily predicted.

2. The key distribution centre sends an encrypted response back to the initiator with a suitable session key for it to use, as well as that session key and an identifier of the initiator encrypted using the master key for the receiver.

   This information for the receiver can only be successfully read by it but will be sent to it by the initiator to prove the initiator's identity. The key distribution centre also includes the same nonce in its response so that the initiator can match the response with its request

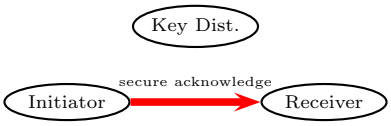
and be sure that the message was not a *replay attack*, where an attacker has duplicated an earlier message (in an attempt to disrupt operation of the system or compromise a key).

3. The initiator forwards the encrypted information it obtained to the receiver to request a secure connection with it. Since this information was encrypted by the key distribution centre for the receiver, only the receiver can decrypt it using its master key.



4. The receiver sends a secure response encrypted using the new session key to acknowledge the request, including a nonce value that it chose and asking the initiator to modify the nonce in some well-known way (such as adding one to it so it changes).

5. The initiator acknowledges the response from the receiver by sending an acknowledgement message encrypted using the session key and containing the modified nonce. Once the receiver obtains its nonce value modified it knows the connection request was genuine and not just a replay attack on it.



Note that a master key is only used by a process with the key distribution centre for securely requesting and transmitting new session keys, it is not used for encrypting large amounts of plaintext, which hinders potential attacks on a master key that rely on a large number of encrypted blocks. Session keys are used rather than master keys for all the encryption of information between processes in the distributed system.

If there is a lot of communication between two processes they might periodically request a new session key from the key distribution centre so as to avoid using the same key for encrypting many plaintext blocks. If two processes already have secure communication a new session key can be arranged between them without having to involve the key distribution centre again. To ensure that a request by one process for a change in session key is genuine (and not a replay attack of a previous request), the initiator of the change can send a nonce value with its encrypted request. The other process sends an encrypted response containing the new suggested session key and a well-known modification to the nonce, along with its own nonce value. When the initiator receives the response it sends an encrypted acknowledgement with a well-known modification to the other's nonce value, proving that it was a genuine request.

Secret keys for symmetric encryption and nonce values to foil replay attacks are typically generated using pseudo-random number generators. It is essential for good security that the sequence of values produced by a random number generator not be easily predicted by a third party. For instance, in 1995 it was announced that the SSL used by some popular browsers could be broken in minutes, not because of the encryption algorithms used by SSL but due to the predictable way the browsers generated random numbers. The `java.util` class `Random` uses a *linear congruential* formula with a 48-bit seed to generate values $x_0, x_1, x_2, \ldots$, which are given by the recurrence relation $x_{n+1} = (ax_n + c) \mod m$ for some carefully chosen values of a, c, and m. Although adequate for many purposes such techniques are not considered *cryptographically strong*, not meeting guidelines for sufficiently unpredictable sequences of random numbers and possibly being vulnerable to cryptanalysis attacks. Hence, the `java.security` package includes the `SecureRandom` class which uses cryptographic algorithms to encrypt a seed and extract a pseudo-random number from the resulting ciphertext. A statement such as:

```
SecureRandom generator = new SecureRandom();
```

uses a random number generator chosen by an available cryptographic security provider, whereas a statement such as:

```
SecureRandom generator=SecureRandom.getInstance("SHA1PRNG");
```

specifies the particular cryptographic algorithm that should be used by the provider. The Java `SecureRandom` class should always be used in place of `Random` for generating confidential random numbers.

It is usually impossible to rely on link encryption when using a mobile device for communication. Even if the telecommunication provider employs link encryption for communication within its own network HTTP requests still traverse the Web where link encryption is not usually used. Hence end-to-end encryption is particularly important. Android includes required support for HTTPS (discussed in Section 4.3) which is adequate for most mobile applications, but if a protocol such as SMS messaging is used instead of HTTP or the Web server receiving the HTTP request is not trusted by the application, then the application must provide its own encryption. Bouncy Castle is a very popular open-source Java package available from http://www.bouncycastle.org/ that provides an alternative JCA cryptographic security provider. It provides implementations of all the main cryptographic algorithms as well as some newer algorithms. Android uses the JCE cryptographic API with the Bouncy Castle security provider, although older versions of Android have a fairly incomplete implementation of the Bouncy Castle API (so the Android-specific *Spongy Castle Cryptography API* might instead be added as a more comprehensive cryptographic library for Android devices prior to version 4.0). The Eclipse project `AndroidAESDemo` demonstrates an Android application that again encrypts SMS messages with AES encryption and CBC mode. The activity `AndroidAESDemoActivity` encrypts the contents of each SMS message before it is sent, and encodes the resulting `byte[]` as a `String` using Base 64 encoding (discussed in Section 3.3) as Android only supports text content in SMS messages, whereas the broadcast receiver `SMSBroadcastReceiver` decodes a `String` SMS message back to a `byte[]` and decrypts it to retrieve the plaintext message contents.

SMS messaging encrypted with AES

```
/**
 * Android Activity that sends SMS messages which it has encrypted using AES and
     Base64 encoded as a String. Note this example uses a hard-coded secret key for
     simplicity
 * @author Andrew Ensor
 */
package aut.hss;
...
public class AndroidAESDemoActivity extends Activity implements
   OnClickListener
{
   private Button sendButton;
   private SMSSentBroadcastReceiver sentBroadcastReceiver;
   private SMSDeliveredBroadcastReceiver deliveredBroadcastReceiver;
   private byte[] keyBytes = { 51, 50, 7, -19, 120, 111, -110, 52, 9, -21, -6, -15,
       -95, 117, 36, -89 }; // the secret key
   private byte[] ivBytes = { 1, -2, 3, -4, 5, -6, 7, -8, 9,-10, 11, -12, 13, -14, 15,
       -16 }; // random array of 16 bytes
   private SecretKeySpec key;
   private IvParameterSpec initVector;
   private final String SMS_SENT_ACTION = "SMS_SENT";
   private final String SMS_DELIVERED_ACTION = "SMS_DELIVERED";

   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState)
   { super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      // obtain reference to send button
```

```
      sendButton = (Button) findViewById(R.id.send_button);
      sendButton.setOnClickListener(this);
      key = new SecretKeySpec(keyBytes, "AES");
      initVector = new IvParameterSpec(ivBytes);
   }
   ...
   // implementation of OnClickListener method
   public void onClick(View view)
   { if (view == sendButton)
      { TextView numberTextView = (TextView)findViewById(R.id.number_text);
         String numberString = numberTextView.getText().toString();
         TextView messageTextView = (TextView)findViewById(R.id.message_text);
         String messageString = messageTextView.getText().toString();
         // send the sms message
         PendingIntent sentPendingIntent = PendingIntent.getBroadcast
            (this, 0, new Intent(SMS_SENT_ACTION), 0);
         PendingIntent deliveredPendingIntent = PendingIntent.getBroadcast(this, 0,
            new Intent(SMS_DELIVERED_ACTION), 0);
         SmsManager smsManager = SmsManager.getDefault();
         String sendString = encryptAndEncodeString(messageString);
         if (sendString != null)
            smsManager.sendTextMessage(numberString, null, sendString,
               sentPendingIntent, deliveredPendingIntent);
      }
   }

   private String encryptAndEncodeString(String message)
   { String errorMessage = null;
      try
      { // create a cipher
         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
         // initialize cipher for encryption
         cipher.init(Cipher.ENCRYPT_MODE, key, initVector);
         // encrypt the plaintext
         byte[] plaintext = message.getBytes();
         byte[] ciphertext = cipher.doFinal(plaintext);
         // base 64 encode the ciphertext as a string
         String encodedString = Base64.encodeToString(ciphertext,
            Base64.DEFAULT);
         return encodedString;
      }
      ...
      Toast toast = Toast.makeText(this, errorMessage,
         Toast.LENGTH_SHORT);
      toast.show();
      return null;
   }
}
```

SMS Broadcast Receiver

```
/**
 * BroadcastReceiver that is notified when an SMS message is received which it Base64
     decodes and decrypts using AES with a hardcoded key. Note this BroadcastReceiver
     is statically registered in AndroidManifest.xml and application requires
     permission android.permission.RECEIVE_SMS @see AndroidAESDemoActivity.java
 */
package aut.hss;
...
public class SMSBroadcastReceiver extends BroadcastReceiver
{
   private byte[] keyBytes = { 51, 50, 7, -19, 120, 111, -110, 52, 9,-21, -6, -15,
       -95, 117, 36, -89 }; // the secret key
```

```
  private byte[] ivBytes = { 1, -2, 3, -4, 5, -6, 7, -8, 9, -10, 11, -12, 13, -14,
      15, -16 }; // random array of 16 bytes
  private SecretKeySpec key;
  private IvParameterSpec initVector;
  public SMSBroadcastReceiver()
  { key = new SecretKeySpec(keyBytes, "AES");
    initVector = new IvParameterSpec(ivBytes);
  }

  public void onReceive(Context context, Intent intent)
  { // obtain the SMS message
    Bundle bundle = intent.getExtras();
    if (bundle != null)
    { Object[] pdus = (Object[]) bundle.get("pdus");
      StringBuilder stringBuilder = new StringBuilder();
      for (int i = 0; i < pdus.length; i++)
      { SmsMessage message = SmsMessage.createFromPdu((byte[]) pdus[i]);
        String senderAddress = message.getDisplayOriginatingAddress();
        String receivedString = message.getDisplayMessageBody();
        String messageString = decodeAndDecryptString(context, receivedString);
        stringBuilder.append("Received Encrypted SMS:\n");
        stringBuilder.append(" Sender: ").append(senderAddress);
        stringBuilder.append(" Message: ").append(messageString);
        stringBuilder.append("\n");
      }
      Toast toast = Toast.makeText(context,
        stringBuilder.toString(), Toast.LENGTH_SHORT);
      toast.show();
    }
    else
    { Toast toast = Toast.makeText(context,
        "Error: no message data received", Toast.LENGTH_SHORT);
      toast.show();
    }
  }

  private String decodeAndDecryptString(Context context,
    String encodedString)
  { String errorMessage = null;
    // base 64 decode the ciphertext as a byte[]
    byte[] ciphertext = Base64.decode(encodedString, Base64.DEFAULT);
    try
    { // create a cipher
      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
      // initialize cipher for encryption
      cipher.init(Cipher.DECRYPT_MODE, key, initVector);
      // decrypt the ciphertext
      byte[] deciphertext = cipher.doFinal(ciphertext);
      return new String(deciphertext);
    }
    ...
  }
}
```

Exercise 1.6 (AES Encryption on a Mobile) *Enhance one of the AES mobile examples so that the sender and receiver of SMS messages can themselves enter a string that is used to determine the 128-bit key for AES encryption and decryption.*

Chapter 2

Public-Key Encryption

2.1 Modular Arithmetic

Besides its importance for AES, the modular arithmetic operations in the ring $\mathbb{Z}_n$ have an important role in public-key encryption. Calculating an arithmetic expression modulo n can be simplified by taking the modulo of each term before performing the arithmetic operations:

$$\begin{aligned}(a+b) \bmod n &= ((a \bmod n) + (b \bmod n)) \bmod n \\ (a-b) \bmod n &= ((a \bmod n) - (b \bmod n)) \bmod n \\ (a \cdot b) \bmod n &= ((a \bmod n) \cdot (b \bmod n)) \bmod n.\end{aligned}$$

Prime numbers turn out to be very useful when performing modular arithmetic. The following fundamental result demonstrates the importance of prime numbers in number theory.

Unique Prime Factorization: Every integer $a > 1$ can be factorized in a unique way as a product of prime numbers:

$$a = p_1^{a_1} p_2^{a_2} \dots p_t^{a_t}$$

where $p_1 < p_2 < \cdots < p_t$ are prime numbers and the exponents $a_1, a_2, \dots, a_t$ are positive integers.

For example:

$$\begin{aligned}21600 &= 2^5 \cdot 3^3 \cdot 5^2 \\ 49000 &= 2^3 \cdot 5^3 \cdot 7^2 \\ 65535 &= 3^1 \cdot 5^1 \cdot 17^1 \cdot 257^1 \\ 65536 &= 2^{16} \\ 65537 &= 65537^1 \qquad \text{as 65537 is itself prime.}\end{aligned}$$

As a consequence, if a prime p divides a product ab then it must be one of the primes in the factorization of the product ab and thus also be a prime in the factorization of either a or of b, so must divide either a or b (or both).

Fermat's Theorem: If p is a prime number and a is a positive integer not divisible by p then $a^{p-1} \bmod p = 1$.

As an example, 7 is a prime number that does not divide 21600 (as 7 does not appear in its prime factorization). Hence $21600^6 \bmod 7 = 1$.

To see why Fermat's Theorem is true consider the numbers $1, 2, \dots, p-1$, multiply each number by a and take the remainder modulo p to get the set:

$$\{a \bmod p, 2a \bmod p, \dots, (p-1)a \bmod p\}.$$

Note that none of the elements in this set are 0, which can be seen by using a proof by contradiction (if $ia \bmod p = 0$ for some i with $1 \le i \le p-1$ then p divides ia, but p does

not divide a so p must divide i, which is not possible). Furthermore, all the elements of this set are distinct from each other, which can also be seen by using a proof by contradiction (if $ia \bmod p = ja \bmod p$ where $1 \leq i < j \leq p-1$ then p must divide $(j-i)a$, but as a is not divisible by p, p must be divide $j-i$, which is not possible as both $i < p$ and $j < p$). Hence this set is just a permutation of $\{1, 2, \ldots, p-1\}$. Multiplying all the elements together therefore gives:

$$\begin{aligned}(a \bmod p) \cdot (2a \bmod p) \cdot \cdots \cdot ((p-1)a \bmod p) &= 1 \cdot 2 \cdot \cdots \cdot (p-1) \\ (a \cdot 2a \cdot \cdots \cdot (p-1)a) \bmod p &= (p-1)! \bmod p \\ a^{p-1}(p-1)! \bmod p &= (p-1)! \bmod p\end{aligned}$$

So p divides $(a^{p-1}-1)(p-1)!$. As p does not divide $(p-1)!$ it must divide $a^{p-1}-1$ which verifies Fermat's Theorem.

Now, if p is an odd prime number then $p-1$ is even and so $p-1 = 2^k q$ for some $k \geq 1$ and odd value of q. Hence for any $a < p$ Fermat's Theorem gives that $a^{2^k q} \bmod p = 1$. As a consequence since p is prime, it can be shown that the values:

$$a^q \bmod p, a^{2q} \bmod p, a^{2^2 q} \bmod p, a^{2^3 q} \bmod p, \ldots, a^{2^k q} \bmod p = 1$$

where each term is the square of the previous modulo p, are either all 1 or else one is equal to $-1 \bmod p = p-1$ and the next is 1. Interestingly, for an odd value of n that is not a prime if a random value of $a < n-1$ is chosen it has probability less than $\frac{1}{4}$ of satisfying this condition. This gives a simple $O(\log_2 n)$ technique for determining whether a number is *probably prime* which can be much more efficient than the brute-force $O(\sqrt{n})$ approach of checking whether any of the numbers $2, 3, \ldots, \sqrt{n}$ divide n. If PROBABLY-PRIME(n) returns FALSE then n is definitely not a prime, whereas if it returns TRUE then the probability that it is a prime is at least $\frac{3}{4}$.

PROBABLY-PRIME(n)

```
1  ▷ determine whether the odd value of n > 2 is probably a prime
2  Find integers k and q for which n − 1 = 2^k q
3  ▷ pick a random integer between 2 and n − 1 exclusive to use in test
4  a ← RANDOM(2, n − 1)
5  current ← a^q mod n
6  if current = 1 then
7          return TRUE                              ▷ n is probably a prime
8  for i ← 1 to k do
9          next ← current^2 mod n
10         if next = 1 then
11                 if current ≠ n − 1 then
12                         return FALSE             ▷ n is definitely not a prime
13                 else
14                         return TRUE              ▷ n is probably a prime
15         current ← next
16 ▷ After end of loop it must be that current ≠ 1
17 return FALSE                                     ▷ n is definitely not a prime
```

MILLER-RABIN(n, s)

```
1  if n = 2 then
2          return TRUE
3  else if n mod 2 = 0 then
4          return FALSE                                          ▷ n is even
5  ▷ test whether n is a prime s times
6  for j ← 1 to s do
7          if not PROBABLY-PRIME(n) then
8                  return FALSE                  ▷ n is definitely not a prime
9  return TRUE          ▷ n is probably a prime with prob at least 1 − 1/4^s
```

The *Miller-Rabin* algorithm simply repeatedly tests PROBABLY-PRIME(n) s times, and will mistakenly claim that an odd value of n is a prime with probability less than $\frac{1}{4^s}$. Thus the

algorithm can be used to determine with any desired probability whether a value of n is a prime, simply by choosing a large enough value of s. In fact, if n is a randomly chosen large number it can be shown that taking $s = 3$ is very unlikely to lead to a false result (the non-prime *Carmichael numbers* such as 561, 1105, 1729 do always get mistaken by the Miller-Rabin algorithm, but they are extremely rare with only 255 such numbers less than 100000000).

If n is a positive integer then the *Euler totient function* $\phi(n)$ is the number of positive integers m less than n that are *relatively prime* to n, meaning that $\gcd(m, n) = 1$. By convention one takes $\phi(1) = 1$.

Euler Totient Function for Small Values of n

n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$
1	1	2	1	3	2	4	2	5	4
6	2	7	6	8	4	9	6	10	4
11	10	12	4	13	12	14	6	15	8
16	8	17	16	18	6	19	18	20	8
21	12	22	10	23	22	24	8	25	20
26	12	27	18	28	12	29	28	30	8

For example, to find $\phi(18)$ one lists the numbers 1 to 17 and removes those that are not relatively prime to 18:

$$\mathbf{1}, \textit{2}, \textit{3}, \textit{4}, \mathbf{5}, \textit{6}, \mathbf{7}, \textit{8}, \textit{9}, \textit{10}, \mathbf{11}, \textit{12}, \mathbf{13}, \textit{14}, \textit{15}, \textit{16}, \mathbf{17}.$$

Only 6 numbers remain and so $\phi(18) = 6$.

If p is a prime then clearly all positive integers less than p are relatively prime to p and so $\phi(p) = p - 1$. Also, it is not difficult to show that if p and q are distinct primes then $\phi(pq) = (p-1)(q-1)$. Determining $\phi(n)$ for most other values of n can be shown to be about as difficult as is determining the unique prime factorization of n, which is usually quite difficult for large values of n.

The following result generalizes Fermat's Theorem for the case when arithmetic is taken modulo a value n that might not be a prime. It can be proved in almost exactly the same way as Fermat's Theorem, but by starting with the $\phi(n)$ numbers that are relatively prime to n.

EULER'S THEOREM: If n and a are positive integers that are relatively prime then $a^{\phi(n)} \bmod n = 1$.

For example, 18 and 35 are relatively prime as $\gcd(18, 35) = 1$ (this can also be seen as their prime factorizations have no primes in common). As $\phi(18) = 6$ Euler's Theorem gives that $35^6 \bmod 18 = 1$.

The following result is also useful for performing modular arithmetic. It states that arithmetic in $\mathbb{Z}_n$ can be performed by using (simpler) arithmetic in $\mathbb{Z}_{n_i}$ for some relatively prime divisors n_i of n.

CHINESE REMAINDER THEOREM: Suppose $n = n_1 n_2 \dots n_k$ where $n_1, n_2, \dots, n_k$ are all relatively prime, and define the function $f\colon \mathbb{Z}_n \to \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ by:

$$f(a) = (a \bmod n_1, a \bmod n_2, \dots, a \bmod n_k).$$

Then f is a bijection for which:

$$\begin{aligned} f(a+b) &= f(a) + f(b) \\ f(a-b) &= f(a) - f(b) \\ f(a \cdot b) &= f(a) \cdot f(b). \end{aligned}$$

where arithmetic in $\mathbb{Z}_n$ is taken modulo n and in $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ is taken modulo n_i in each coordinate. The inverse $f^{-1}\colon \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k} \to \mathbb{Z}_n$ is given by:

$$f^{-1}(a_1, a_2, \dots, a_k) = (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \bmod n,$$

where each $c_i = (n/n_i) \cdot d_i$ and d_i is the value in $\mathbb{Z}_{n_i}$ for which $(n/n_i) \cdot d_i \bmod n_i = 1$ (d_i can be obtained by the Euclidean algorithm).

To illustrate the Chinese Remainder Theorem, suppose a is a value for which $a \bmod 5 = 3$ and $a \bmod 12 = 7$. To determine the value of a (which is unique modulo $5 \cdot 12 = 60$) one approach would be to make a table of all values between 0 and 59 where the rows have the same remainder modulo 5 and the columns have the same remainder 12. Then the correct value of a can be read off the row with remainder 3 and column with remainder 7, giving $a = 43$. Alternatively, as $n_1 = 5$ and $n_2 = 12$ are relatively prime the Chinese Remainder Theorem can

Values of a between 0 *and* 59 *with remainders modulo* 5 *and modulo* 12

Remainders	0	1	2	3	4	5	6	7	8	9	10	11
0	0	25	50	15	40	5	30	55	20	45	10	35
1	36	1	26	51	16	41	6	31	56	21	46	11
2	12	37	2	27	52	17	42	7	32	57	22	47
3	48	13	38	3	28	53	18	43	8	33	58	23
4	24	49	14	39	4	29	54	19	44	9	34	59

be used to find $a = f^{-1}(3, 7)$. Let $n = n_1 \cdot n_2 = 60$, so that $n/n_1 = 12$ and $n/n_2 = 5$. Then d_1 is the value for which $12 \cdot d_1 \bmod 5 = 1$, which by the Euclidean algorithm can be found to be $d_1 = 3$. Similarly, d_2 is the value for which $5 \cdot d_2 \bmod 12 = 1$, which is $d_2 = 5$. Then $c_1 = (n/n_1) \cdot d_1 = 12 \cdot 3 = 36$ and $c_2 = (n/n_2) \cdot d_2 = 5 \cdot 5 = 25$. Now any value of $f^{-1}(a_1, a_2)$ can be found, such as $a = f^{-1}(3, 7) = (3 \cdot 36 + 7 \cdot 25) \bmod 60 = 43$.

Exercise 2.1 (Random Prime Generator) *Prepare a program that can generate a cryptographically strong large random prime number (for a very large number you might like to make use of the* `BigInteger` *class that is in the* `java.math` *package).*

2.2 RSA Algorithm

The concept of public-key cryptography was introduced as a way to distribute secret keys without resorting to physical distribution, and as a way to authenticate the sender of a message. Public-key cryptography uses two keys, a *public key* which is usually publicly available and a *private key* which must be kept confidential. Encrypting plaintext using one of the keys results in ciphertext which should only be able to be decrypted with the other key. Hence the holder of the private key should be the only one that can decrypt messages encoded by anyone using the public key, enabling others to encrypt messages that they send to the holder. The holder should also be the only one that can encrypt messages that get decrypted by anyone using the public key, enabling others to authenticate that a message was produced by the holder (provided they are certain of the origin of the public key).

One of the first public-key ciphers and currently the most-popular public key block cipher was developed by Rivest, Shamir, Adleman and known as the *RSA algorithm.* The public and private keys are calculated with the following steps:

1. two large prime numbers p and q are selected at random, typically each at least 512 bits,
2. the product $n = pq$ is calculated, allowing encryption of any plaintext block whose value is in $\mathbb{Z}_n$ (typically at least 1024 bits),
3. a small odd integer e that is relatively prime to $\phi(n) = (p-1)(q-1)$ is selected, a common choice is $e = 65537$ so long as $\gcd(\phi(n), e) = 1$,
4. the multiplicative inverse d of e in $\mathbb{Z}_{\phi(n)}$ is calculated using the Euclidean algorithm (which is the value s calculated by EXTENDED-EUCLID$(e, \phi(n))$),
5. the public key is the pair (e, n) which is published,
6. the private key is the pair (d, n) which is kept secret.

A plaintext block P is considered a value in $\mathbb{Z}_n$. Encryption of P using the public key into a ciphertext value C is given by:

$$C = P^e \bmod n,$$

EXTENDED-EUCLID(m, n)

```
1  ▷ determine d = gcd(m, n) and s, t for which gcd(m, n) = s · m + t · n
2  if n = 0 then
3          return (m, 1, 0)
4  (d′, s′, t′) ← EXTENDED-EUCLID(n, m mod n)
5  q ← ⌊m/n⌋                                   ▷ integer division m/n
6  (d, s, t) ← (d′, t′, s′ − qt′)
7  return (d, s, t)
```

whereas for authentication the private key d would instead be used for encryption. Decryption is then achieved by the same technique but with the other key:

$$P = C^d \bmod n.$$

To verify that encryption and decryption are indeed inverse operations, it must be shown that $(P^e \bmod n)^d \bmod n = P$ for any possible value P in $\mathbb{Z}_n$. Note that $ed \bmod \phi(n) = 1$ and $\phi(n) = (p-1)(q-1)$ as p and q are both primes, so that $ed = 1 + k(p-1)(q-1)$ for some multiple k. Thus:

$$(P^e \bmod n)^d \bmod n = P^{ed} \bmod n = P^{1+k(p-1)(q-1)} \bmod n.$$

Now, if P is not divisible by p then $P^{p-1} \bmod p = 1$ by Fermat's Theorem, so:

$$\begin{aligned} P^{1+k(p-1)(q-1)} \bmod p &= P \cdot \left(P^{p-1}\right)^{k(q-1)} \bmod p \\ &= P \cdot (1)^{k(q-1)} \bmod p \\ &= P \bmod p. \end{aligned}$$

If instead P is divisible by p then too $P^{1+k(p-1)(q-1)} \bmod p = 0 = P \bmod p$. Similarly, $P^{1+k(p-1)(q-1)} \bmod q = P \bmod q$, so the Chinese Remainder Theorem can be applied to show that $P^{1+k(p-1)(q-1)} \bmod n = P \bmod n$. This verifies that RSA decryption is the reverse of RSA encryption.

The exponentiation operations P^e and C^d used in encryption and decryption can be performed in $\Theta(\log_2 e)$ and $\Theta(\log_2 d)$ respectively. For instance $P^e \bmod n$ can be found by calculating:

$$P, P^2 \bmod n, P^4 \bmod n, P^8 \bmod n, P^{16} \bmod n, \ldots$$

where each term is the square modulo n of the previous term. The algorithm MODULAR-EXPONENTIATION demonstrates how this can be achieved using the binary representation of an exponent b. For example, if $b = 22$ which has binary representation 10110, then after the first iteration of the loop $b_4 = 1$ and so $t = a \cdot 1 = a$. During the next iteration $b_3 = 0$ and so t is just squared giving $t = a^2$. During the third iteration $b_2 = 1$ and so t is squared and then multiplied by a giving $t = a^5$. During the fourth iteration $b_1 = 1$ and so t is again squared and multiplied by a, resulting in $t = a^{11}$. In the final iteration $b_0 = 0$ and so the result is just squared, giving $t = a^{22}$. Actually, since the exponent d is probably very large a

MODULAR-EXPONENTIATION(a, b, n)

```
1  ▷ determine a^b mod n in order Θ(log_2 b) where b = b_{k−1} b_{k−2} ... b_0 in binary
2  t ← 1
3  for i ← k−1 downto 0 do
4          t ← t × t mod n
5          if b_i = 1 then
6                  t ← (t × a) mod n
7  return t
```

more efficient way of calculating C^d during decryption can be advantageous. Using Fermat's Theorem, $C^d \bmod p = C^{d \bmod (p-1)} \bmod p$ and $C^d \bmod q = C^{d \bmod (q-1)} \bmod q$. So $C^d \bmod p$ and $C^d \bmod q$ can be found relatively quickly. Then as $n = pq$ and p and q are relatively prime,

the Chinese Remainder Theorem can be applied to obtain $C^d \bmod n$. This way is about four times faster than simply using MODULAR-EXPONENTIATION to calculate $C^d \bmod n$.

It may appear surprising that n can be included as part of the public key, as if a cryptanalyst were to find the primes p and q with $n = pq$, or even just calculate $\phi(n)$ then the private key would be obtained. However, obtaining the prime factorization for large values of n is well-known to be difficult, with no efficient algorithm yet discovered. Also, it has been shown that calculating $\phi(n)$ is equivalent to factorizing $n = pq$. Hence the security of the RSA algorithm is based on the difficulty of factorizing a sufficiently large value of n.

Cryptanalysis attacks on RSA based on factorizing n have improved since the release of RSA. When RSA was first published in 1977 a challenge was made to break a particular ciphertext encrypted using a 428 bit value for n, which at the time was presumed would take 40 quadrillion years by brute force. But in 1994, after only eight months work the ciphertext was successfully broken, and since then factorization techniques have improved substantially, so that by 2005 a 663 bit value of n was no longer considered secure. Hence larger values (currently between 1024 bit and 2048 bit) are now used for RSA encryption. However, the drawback is that encryption and decryption calculations are longer, slowing the algorithm.

There are several other interesting cryptanalysis attacks on RSA. If a particularly small value of e is chosen and if the same plaintext is encrypted for that number of users each with different values of n then the plaintext can be easily obtained. For instance, if $e = 3$ (which was once commonly used) and the same plaintext P is encrypted with n_1, n_2, n_3 and intercepted by a cryptanalyst, then the values $P^3 \bmod n_1$, $P^3 \bmod n_1$, $P^3 \bmod n_1$ are obtained. As n_1, n_2, n_3 are probably relatively prime with each other, the Chinese Remainder Theorem can then be used to obtain $P^3 \bmod n_1 n_2 n_3$. But $P < n_1$, $P < n_2$, $P < n_3$, so P^3 has been found. Taking its cube root then gives the original plaintext P.

Another approach that demonstrates the ingenuity of cryptanalysis attacks is known as a *timing attack*, which relies on the time taken by an algorithm such as RSA to decrypt ciphertext. As the time taken by each iteration of the MODULAR-EXPONENTIATION algorithm depends on whether the bit d_i of the private key d is 0 or 1, by carefully choosing ciphertext, a cryptanalyst can determine bit by bit the value of d. To foil a timing attack, RSA algorithms usually include *blinding*, where a secret random number r is chosen, encrypted and multiplied by the ciphertext C before it gets decrypted. This ensures that the ciphertext actually being decrypted is not known to the cryptanalyst. The decrypted result is then multiplied by the multiplicative inverse of r in $\mathbb{Z}_n$ to give the original plaintext.

RSA has further properties that can be exploited by a cryptanalyst. It is easily seen that the ciphertext for $P_1 \cdot P_2$ is the same as the product of the ciphertext for P_1 with the ciphertext for P_2. As a consequence, if a cryptanalyst intercepts a ciphertext C and is then allowed to obtain the plaintext for certain chosen ciphertext, then the plaintext for the intercepted ciphertext C can be quickly obtained. To counter this attack, RSA algorithms use *optimal asymmetric encryption padding* (OAEP) where all plaintext are randomly padded before they get encrypted.

Due to the length of the key needed to ensure security of RSA, and the resulting long calculations, RSA is not nearly as efficient as symmetric cipher algorithms such as AES for encrypting and decrypting large quantities of data. For this reason RSA is typically used only for the initial exchange of a secret key after which a symmetric cipher is used for further encryption and decryption.

The class `RSADecrypter` demonstrates how a pair of keys can be generated in Java using the `KeyPairGenerator` class:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024); // size of key in bits
KeyPair keyPair = kpg.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
```

From the two keys their modulus n and exponent e or d can be obtained by typecasting them to `RSAPublicKey` and `RSAPrivateKey` respectively. Once the key pair has been generated the example sends the public key via a TCP socket connection to another class called `RSAEncrypter`

(either a Java or a C# class) that is listening for socket connections. The `RSAEncrypter` obtains the modulus n and the public exponent e and uses them to encrypt a short plaintext using RSA. The encrypted ciphertext is then passed back to `RSADecrypter` where it is decrypted using the private key. In a more-realistic application this plaintext might typically be a secret key for symmetric encryption, and the public key would be passed using a standard format such as the X.509 standard (discussed in Section 3.4). Furthermore, this example does not use OAEP which is supported in both Java and C#.

RSA encryption is performed in C# using the `RSACryptoServiceProvider` class (instead of the usual C# `CipherStream`). Note if a key of length exactly a power of two is used then the `RSACryptoServiceProvider` cipher appears to add an extra byte to the encrypted ciphertext, which is inconsistent with the length of the key, resulting in a `BadPaddingException` in `RSADecrypter`. This small incompatibility can be avoided by not using keys of lengths exactly 512 or 1024 in the C# version.

RSA ENCRYPTION

```
/**
   A class that demonstrates how RSA encryption can be used to obtain
   RSA encrypted information from an RSAEncrypter server.
   To run first start RSAEncrypter.
   @author Andrew Ensor
*/
...
public class RSADecrypter
{
   public static final String HOST_NAME = "localhost";
   public static final int HOST_PORT = 8888; // host port number
   private KeyPair keyPair;

   public RSADecrypter()
   { // generate an RSA public and private key pair
      try
      { KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
         kpg.initialize(1024); // size of key in bits
         keyPair = kpg.generateKeyPair();
      }
      catch (NoSuchAlgorithmException e)
      { System.err.println("Encryption algorithm not available: "+e);
      }
   }

   private void obtainBytes()
   { // open TCP connection to RSAEncrypter server on specified port
      Socket socket = null;
      OutputStream os = null;
      InputStream is = null;
      try
      { socket = new Socket(HOST_NAME, HOST_PORT);
         os = socket.getOutputStream();
         // send the public key modulus and public exponent to host
         RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
         System.out.println("Sending public key");
         byte[] modulus = publicKey.getModulus().toByteArray();
         byte[] exponent = publicKey.getPublicExponent().toByteArray();
         os.write(modulus.length); // write number of bytes in modulus
         os.write(modulus);
         os.write(exponent.length);//write number of bytes in exponent
         os.write(exponent);
         os.flush();
         // obtain the encrypted bytes
         is = socket.getInputStream();
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
        int data = is.read();
        while (data != -1) // end of stream encountered
        { baos.write(data);
          data = is.read();
        }
\end{verbatim}\hfill \emph{cont-}\end{program}\end{figure*}%
\begin{figure*}\begin{program}\emph{-cont}\begin{verbatim}
        byte[] ciphertext = baos.toByteArray();
        // decrypt the bytes using the private key
        PrivateKey privateKey = keyPair.getPrivate();
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] plaintext = cipher.doFinal(ciphertext);
        // display the plaintext
        System.out.print("Deciphered plaintext:");
        for (int i=0; i<plaintext.length; i++)
          System.out.print(" " + plaintext[i]);
        System.out.println();
      }
      catch (IOException e)
      { System.err.println("IOException in decrypter: " + e);
      }
      catch (NoSuchAlgorithmException e)
      { System.err.println("Encryption algorithm not available: "+e);
      }
      catch (NoSuchPaddingException e)
      { System.err.println("Padding scheme not available: " + e);
      }
      catch (InvalidKeyException e)
      { System.err.println("Invalid key: " + e);
      }
      catch (IllegalBlockSizeException e)
      { System.err.println("Cannot pad plaintext: " + e);
      }
      catch (BadPaddingException e)
      { System.err.println("Exception with padding: " + e);
      }
      finally
      { try
        { if (os != null) os.close();
          if (is != null) is.close();
          if (socket != null) socket.close();
        }
        catch (IOException e)
        { System.err.println("Failed to close streams: " + e);
        }
      }
   }

   public static void main(String[] args)
   { RSADecrypter decrypter = new RSADecrypter();
     decrypter.obtainBytes();
   }
}
```

RSA encryption

```
/**
   A class that demonstrates how RSA encryption can be used to encrypt
   RSA encrypted information to an RSADecrypter client
   @see RSADecrypter.java
*/
...
```

```
public class RSAEncrypter
{
   public static final int PORT = 8888; // some unused port number
   private final byte[] plaintext = {1, 2, 3, 4, 5, 6, 7}; // secret
   ...
   // inner class that represents a single connection handler
   private class ConnectionHandler implements Runnable
   {
      private Socket socket; // socket for client/server communication

      public ConnectionHandler(Socket socket)
      { this.socket = socket;
      }

      public void run()
      { // handle the TCP connection with an RSADecrypter client
         InputStream is = null;
         OutputStream os = null;
         try
         { // obtain public key modulus and public exponent from client
            is = socket.getInputStream();
            int modulusLength = is.read(); // get num bytes in modulus
            byte[] modulus = new byte[modulusLength];
            for (int i=0; i<modulusLength; i++)
               modulus[i] = (byte)is.read();
            int exponentLength = is.read();//get num bytes in exponent
            byte[] exponent = new byte[exponentLength];
            for (int i=0; i<exponentLength; i++)
               exponent[i] = (byte)is.read();
            RSAPublicKeySpec publicKeySpec = new RSAPublicKeySpec
               (new BigInteger(modulus), new BigInteger(exponent));
            // create public key from the modulus and public exponent
            KeyFactory keyFactory = KeyFactory.getInstance("RSA");
            PublicKey publicKey
               = keyFactory.generatePublic(publicKeySpec);
            // use the public key to encrypt the information
            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.ENCRYPT_MODE, publicKey);
            byte[] ciphertext = cipher.doFinal(plaintext);
            // send the encrypted bytes
            os = socket.getOutputStream();
            os.write(ciphertext);
         }
         ...
      }
   }
}
```

RSA encryption in C#

```
/*
   A C# class that demonstrates how RSA encryption can be used to
   obtain RSA encrypted information from an RSAEncrypter server.
   Note this example presumes that the modulus length can be
   represented as a byte (an int would be preferable but then the
   endian order needs to be considered), and that the modulus is a
   positive integer in twos-complement notation (so most significant
   bit is zero).
*/
...
public class RSADecrypter
{
   public const String HOST_NAME = "localhost";
```

```
   public const int HOST_PORT = 8888; // host port number
   private RSAParameters keyPair;

   public RSADecrypter()
   { // generate an RSA public and private key pair using 1024 bit key
      RSACryptoServiceProvider cipher
         = new RSACryptoServiceProvider(1024);
      keyPair = cipher.ExportParameters(true);
   }

   private void ObtainBytes()
   { // open TCP connection to RSAEncrypter server on specified port
      TcpClient client = null;
      try
      { client = new TcpClient();
         client.Connect(HOST_NAME, HOST_PORT);
      }
      catch (SocketException e)
      { Console.WriteLine("Client could not make connection: " + e);
         System.Environment.Exit(System.Environment.ExitCode);
      }
      NetworkStream stream = null;
      BinaryWriter bw = null;
      BinaryReader br = null;
      try
      { stream = client.GetStream();
         bw = new BinaryWriter(stream);
         // send the public key modulus and public exponent to host
         Console.WriteLine("Sending public key");
         byte[] modulus = keyPair.Modulus;
         byte[] exponent = keyPair.Exponent;
         if ((modulus[0] & 0x80) != 0)
         { // modulus needs a 00 prepended to put into twos complement
            byte[] twosComplement = new byte[modulus.Length+1];
            twosComplement[0] = 0;
            Array.Copy(modulus, 0, twosComplement, 1, modulus.Length);
            modulus = twosComplement;
         }
\end{verbatim}\hfill \emph{cont-}\end{program}\end{figure*}%
\begin{figure*}\begin{program}\emph{-cont}\begin{verbatim}
         bw.Write((byte)modulus.Length); // write num of modulus bytes
         bw.Write(modulus);
         bw.Write((byte)exponent.Length);//write num of exponent bytes
         bw.Write(exponent);
         bw.Flush();
         // obtain the encrypted bytes
         br = new BinaryReader(stream);
         MemoryStream memoryStream = new MemoryStream();
         BinaryWriter msbw = new BinaryWriter(memoryStream);
         byte[] data = br.ReadBytes(0);
         while (data.Length>0) // end of stream encountered
         { msbw.Write(data);
            data = br.ReadBytes(0);
         }
         stream.CopyTo(memoryStream);
         byte[] ciphertext = memoryStream.ToArray();
         // decrypt the bytes using private key without OAEP padding
         RSACryptoServiceProvider cipher
            = new RSACryptoServiceProvider();
         cipher.ImportParameters(keyPair);
         byte[] plaintext = cipher.Decrypt(ciphertext, true);
         // display the plaintext
         Console.Write("Deciphered plaintext:");
```

```
        for (int i=0; i<plaintext.Length; i++)
           Console.Write(" " + plaintext[i]);
        Console.WriteLine();
        cipher.Dispose();

     }
     catch (SocketException e)
     { Console.WriteLine("Client error: " + e);
     }
     finally
     { try
        { if (br != null) br.Close();
           if (bw != null) bw.Close();
           if (stream != null) stream.Close();
           if (client != null) client.Close();
        }
        catch (SocketException e)
        { Console.WriteLine("Failed to close streams: " + e);
        }
     }
  }

  public static void Main(string[] args)
  { RSADecrypter decrypter = new RSADecrypter();
     decrypter.ObtainBytes();
  }
}
```

RSA encryption in .NET

```
/*
   A C# class that demonstrates how RSA encryption can be used in .NET
   to encrypt RSA encrypted information to an RSADecrypter client.
*/
...
public class RSAEncrypter
{
   ...
   // inner class that represents a single connection handler
   private class ConnectionHandler
   {
      ...
      public void Run() // can call this method anything in C#
      { // handle the TCP connection with an RSADecrypter client
         NetworkStream stream = new NetworkStream(socket);
         BinaryReader br = null;
         BinaryWriter bw = null;
         try
         { // obtain public key modulus and public expon from client
            br = new BinaryReader(stream);
            int modulusLength=br.ReadByte();//get num bytes in modulus
            byte[] modulus = new byte[modulusLength];
            for (int i=0; i<modulusLength; i++)
               modulus[i] = br.ReadByte();
            if (modulus[0] == 0)
            { // an additional 00 byte has probably been appended
               // to make modulus twos complement
               byte[] unsigned = new byte[modulus.Length-1];
               Array.Copy(modulus, 1, unsigned, 0, modulus.Length-1);
               modulus = unsigned;
            }
            int exponentLength=br.ReadByte();//get num bytes in expon
            byte[] exponent = new byte[exponentLength];
```

```
            for (int i=0; i<exponentLength; i++)
               exponent[i] = br.ReadByte();
            RSAParameters publicKeySpec = new RSAParameters();
            publicKeySpec.Modulus = modulus;
            publicKeySpec.Exponent = exponent;
            // create public key from the modulus and public exponent
            RSACryptoServiceProvider cipher
               = new RSACryptoServiceProvider();
            cipher.ImportParameters(publicKeySpec);
            // use the public key to encrypt info without OAEP padding
            byte[] ciphertext = cipher.Encrypt(plaintext, true);
            // send the encrypted bytes
            bw = new BinaryWriter(stream);
            bw.Write(ciphertext);
            cipher.Dispose();
         }
         ...
      }
   }
}
```

Exercise 2.2 (Implementation of RSA) *Prepare a program that provides an implementation of the RSA public-key block cipher.*

2.3 Key Exchange

The performance of public-key encryption algorithms can be up to 1000 times slower than symmetric encryption, which limits their suitability to the exchange of brief messages such as secret keys for symmetric encryption. The classes `RSADecrypter` and `RSAEncrypter` from Section 2.2 demonstrate how an asymmetric cipher such as RSA can be used to pass a secret key safely encrypted between two processes, which can then be used for the encryption of further messages.

The *Diffie-Hellman key exchange* algorithm was the first published public-key algorithm and was devised so that two processes could use insecure communication to determine a common secret key. This algorithm starts with two public values that are pre-computed, a large prime p and an integer value a with $1 < a < p$ for which:

$$a, a^2 \bmod p, a^3 \bmod p, \ldots, a^{p-1} \bmod p = 1.$$

are all distinct (so they must just be a permutation of $\{1, 2, 3, \ldots, p-1\}$) in which case a is called a *primitive root* of p. Once a primitive root a of a prime p has been found, for any integer b there is a unique power $i < p$ for which $b \bmod p = a^i \bmod p$, and i is called the *discrete logarithm* or *index* of b modulo p for the base a.

For example, taking $p = 7$ the value $a = 2$ is not a primitive root of 7 since in $\mathbb{Z}_7$ one has:

$$2^1 = 2, 2^2 = 4, 2^3 = 1, 2^4 = 2, 2^5 = 4, 2^6 = 1,$$

which are not all distinct, whereas the value $a = 3$ is a primitive root as:

$$3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1,$$

which are all distinct. Thus the discrete logarithm of an integer such as $b = 1763$ modulo $p = 7$ for the base $a = 3$ is found by $b \bmod 7 = 6 = 3^3$, and so $i = 3$.

The security of the Diffie-Hellman key exchange relies on the difficulty of computing discrete logarithms for large primes p, much as RSA relies on the difficulty of computing prime factorizations for large integers n. When two processes with the same base a and prime p want to exchange a secret key each selects its own secure random number $x < p$ as its private key and calculates a public key $y = a^x \bmod p$, which is communicated to the other process. When each process receives the public key $y' = a^{x'} \bmod p$ of the other process (but not knowing the other private key x'), it calculates the secret key K:

$$K = (y')^x \bmod p.$$

Note that this is the same key as that calculated by the other process as:

$$\begin{aligned}(y')^x \bmod p &= \left(a^{x'} \bmod p\right)^x \bmod p \\ &= a^{x \cdot x'} \bmod p \\ &= (a^x \bmod p)^{x'} \bmod p \\ &= y^{x'} \bmod p,\end{aligned}$$

but a cryptanalyst cannot calculate K without knowing one of the values x or x'. Obtaining these values would require calculating the discrete logarithm of an intercepted y or y' value modulo p to the base a, which is computationally difficult.

JCA supports Diffie-Hellman key exchange using the algorithm name `DH` and the following steps:

Generate the Public Prime and Primitive Root New values of p and a can be generated for a Diffie-Hellman key exchange using an `AlgorithmParameterGenerator`:

```
AlgorithmParameterGenerator apg
   = AlgorithmParameterGenerator.getInstance("DH");
apg.init(512); // 512-bit prime
DHParameterSpec dhSpec = (DHParameterSpec)
   apg.generateParameters().getParameterSpec
   (DHParameterSpec.class);
```

or else existing values of p and a can be passed directly:

```
DHParameterSpec dhSpec = new DHParameterSpec(p, a);
```

Create Public and Private Keys Values of x and y can be generated by each process by passing a `KeyPairGenerator` the p and a parameters:

```
KeyPairGenerator kpg=KeyPairGenerator.getInstance("DH");
kpg.initialize(dhSpec);
KeyPair keyPair = kpg.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
```

From the two keys their public value y or private value x can be obtained by typecasting the keys to `DHPublicKey` or `DHPrivateKey` respectively.

Swap Public Keys The processes involved in the key exchange swap their public values of y. This can be accomplished by passing the raw values of y (along with p and a if not already publicly available):

```
BigInteger yValue = ((DHPublicKey)publicKey).getY();
... // send yValue
```

and using a `DHPublicKeySpec` with a `KeyFactory` to reassemble the public key at the other end:

```
BigInteger otherYValue = ...; // receive
DHPublicKeySpec otherPublicKeySpec
   = new DHPublicKeySpec(otherYValue, pValue, aValue);
KeyFactory keyFactory = KeyFactory.getInstance("DH");
PublicKey otherPublicKey
   = keyFactory.generatePublic(otherPublicKeySpec);
```

A preferred way to accomplish this is to send the public key in the standardized X.509 key exchange format:

```
byte[] publicKeyBytes = publicKey.getEncoded();
... // send publicKeyBytes
```

and use an `X509EncodedKeySpec` with a `KeyFactory` to reassemble the public key at the other end:

```
byte[] otherPublicKeyBytes = ...; // receive
X509EncodedKeySpec otherPublicKeySpec
```


```
   = new X509EncodedKeySpec(otherPublicKeyBytes);
KeyFactory keyFactory = KeyFactory.getInstance("DH");
PublicKey otherPublicKey
   = keyFactory.generatePublic(otherPublicKeySpec);
```

Generate Secret Key The private key is used to initialize a `KeyAgreement` and then its `doPhase` method is passed the swapped public key(s) (with a boolean to indicate which is the last public key obtained):

```
KeyAgreement ka = KeyAgreement.getInstance("DH");
ka.init(privateKey); // initialize with own private key
ka.doPhase(otherPublicKey, true);
```

Then the `KeyAgreement` method `generateSecret` can be used to generate bytes for a secret key (or for a specific algorithm such as `DES` or `DESede`):

```
byte[] key = ka.generateSecret();
```

However, if the communication channel between two processes can be compromised so that messages can be modified then the Diffie-Hellman key exchange is susceptible to a *man-in-the-middle attack*. With this attack an adversary generates its own key pair and intercepts all messages between the two processes. The attacker starts by replacing the public key of each with its own public key during the key exchange. Then each process will generate a secret key in common with the attacker rather than with each other. The attacker then decrypts messages from either process using one secret key and can re-encrypt them (possibly modified) for the other process using the other secret key. If both ends are unaware that their messages have been modified then their encrypted communication will be compromised.

To counter a man-in-the-middle attack it is important that the recipient of a public key be certain that it is an authentic key for the intended process. Merely broadcasting a public key is not secure as anyone could forge a broadcast, sending a public key generated by themselves and claiming to be another process. The forger would then receive messages encrypted for it that were intended for the genuine process until the forgery were detected and all other processes notified. Instead, the public keys could be made globally available from a secure public directory, which would require a man-in-the-middle attack to intercept messages to and from the directory (which might be able to be made physically secure) as well as eavesdrop on communication between processes.

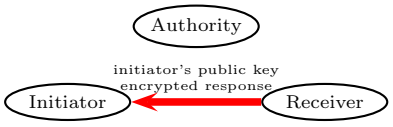
A more secure approach is to use a *public-key authority* which controls access to the public keys in a system. A central authority holds a public key for each process, which is delivered to the authority via some secure means (such as physically), and in exchange the process is given a public key for its communication with the authority. When an initiating process wants to exchange its public key with that of another process it can use the following steps:

1. The initiator sends a request encrypted using the public key of the authority, requesting the public key for the receiving process, and including a timestamp with the request. The public-key authority responds with a message encrypted using its private key, including the public key for the receiver and the original request with the timestamp. Once the response has been decrypted by the initiator using the public key of the authority it is assured that the message was encrypted by the authority and that its original request was received by the authority untampered.

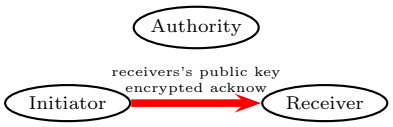
2. The initiator then uses the public key for the receiver to encrypt an identifier for itself and a nonce, which it sends to the receiver, requesting an exchange. It is assured that only the receiver can decrypt the request to obtain its nonce.

3. Upon receiving the request for an exchange, the receiver repeats the same process with the public-key authority to obtain a public key for the initiator. At this stage the initiator and the receiver have securely obtained each other's public key. To assure both ends that they are communicating with each other two further steps are included.

4. The receiver sends a response to the initiator encrypted using the initiator's public key, including the initiator's nonce as well as its own nonce. When the initiator decrypts the response and finds its own nonce it is assured that it is actually communicating with the intended receiver.



5. The initiator acknowledges the response from the receiver by sending an acknowledgement message encrypted using the public key of the receiver, including the receiver's nonce. When the receiver decrypts the acknowledgement and finds its own nonce it is assured that it is actually communicating with the intended initiator.



One drawback of using a central public-key authority is that it can become a bottleneck in a large system, hindering scalability. Also, if the authority were to temporarily fail then no keys would be available. An alternative approach is to use encrypted certificates. A *certificate* holds an identifier for a process and a public key offered by it, prepared by a trusted *certificate authority* and encrypted by the authority using a private key. The certificate authority makes its corresponding public key available so that any process can decrypt the certificate and obtain the public key of the process, but since its private key is confidential no one else can forge a certificate by that authority. So long as each process trusts the certificate authority and is certain that it has the correct public key for the authority, then two processes can exchange public keys simply by sending each other a certificate for itself issued by the authority. Thus certificates can be forwarded around the system even when the authority might be temporarily unavailable. A timestamp is usually included as part of the encrypted certificate so that obsolete or expired certificates can be ignored. Certificates are discussed further in Section 3.4.

Once two processes have successfully exchanged public keys (such as via a public-key authority or by exchanging certificates) the following steps can be followed to protect against attacks before a secret key is exchanged:

1. First, the initiator and receiver assure that they are communicating with each other by exchanging nonce values. The initiator starts by sending a request encrypted using the public key of the receiver, including an identifier for itself and a nonce.
2. The receiver replies with a response encrypted using the public key of the initiator, including the nonce of the initiator as well as its own nonce. This assures the initiator that it is communicating with the receiver.
3. The initiator then sends an acknowledgement encrypted using the public key of the receiver, including the nonce of the receiver. This assures the receiver that it is communicating with the initiator.
4. One end then generates a secret key. Rather than just encrypting it once using the public key of the other process (so that only that process can decrypt it), the encrypted key is also further encrypted using the private key of the process that generated it. This assures the other end that this process was the process that created the key, and so is not susceptible to a man-in-the-middle attack.

Exercise 2.3 (Diffie-Hellman Key Exchange) *Prepare a program that uses the Diffie-Hellman key exchange algorithm over a TCP connection to exchange a secret key.*

2.4 Elliptic Curve Cryptography

Due to recent progress in calculating prime factorizations RSA public keys need to be at least 1024 bits to provide adequate security. *Elliptic Curve Cryptography* (ECC) is a promising alternative to RSA for public-key encryption, allowing a much shorter key to be used with far less computational overhead, yet providing the same level of security as RSA against a cryptanalysis attack.

For instance, in order to provide roughly the same level of security as a 128-bit AES key, RSA requires a 3072-bit key, which places quite a computational burden on any devices using RSA. Worse still, to be equivalent to a 256-bit AES key, RSA requires a 15360-bit key, which is infeasible on limited devices such as mobile phones. ECC however requires just double the number of bits than an AES key, making it particularly attractive for public-key encryption on limited devices. It has begun to challenge RSA and Diffie-Hellman key exchange as the preferred public-key cryptographic algorithm, since the difficulty of cryptanalysis against ECC gets harder for longer keys much faster than for either RSA or Diffie-Hellman.

Comparable Key Sizes

AES	*RSA*	*ECC*
—	512	112
—	1024	160
—	2048	224
128	3072	256
192	7680	384
256	15360	512

An *elliptic curve* over a field consists of the set of points (x, y) where x and y are elements of the field that satisfy an equation of the form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

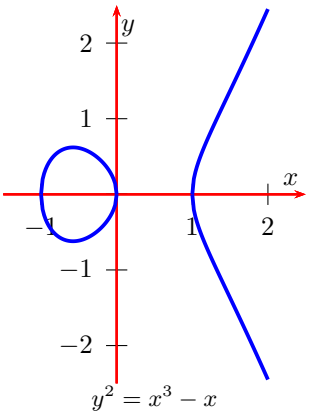
where the coefficients a_1, a_3, a_2, a_4, a_6 are also elements of the field.

If the number of elements in the field is not divisible by 2 nor 3 (such as the field $\mathbb{R}$ of all real numbers or the finite field $\mathbb{Z}_p$ for a prime $p > 3$) then using a suitable affine transformation results in the simpler equation:

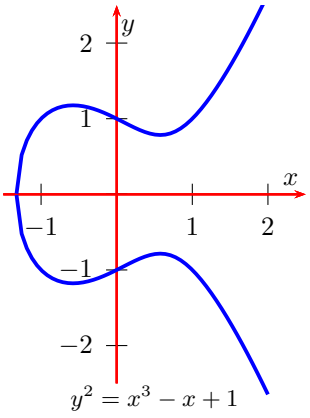
$$y^2 = x^3 + ax + b.$$

If instead the number of elements is divisible by 2 but not by 3 (such as the field $GF(2^m)$) then the equation is more conveniently expressed in the form $y^2 + xy = x^3 + ax^2 + b$.

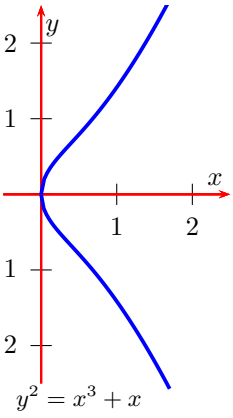
The *discriminant* $\Delta = -16\left(4a^3 + 27b^2\right)$ for an elliptic curve $y^2 = x^3 + ax + b$ is a quantity analogous to $\Delta = b^2 - 4ac$ for a quadratic curve $y = ax^2 + bx + c$, that determines the number of values of x for which $y = 0$. If $\Delta \neq 0$ then there are three distinct (possibly complex) solutions. For example, the elliptic curves $y^2 = x^3 - x$, $y^2 = x^3 - x + 1$, $y^2 = x^3 + x$ all have non-zero discriminant, and so $x^3 + ax + b = 0$ has three distinct solutions for x (the first curve has roots $x = -1, 0, 1$, the second has roots $x \approx -1.3247, 0.6624 + 0.5623i, 0.6624 - 0.5623i$, and the third has $x = 0, i, -i$).



$y^2 = x^3 - x$



$y^2 = x^3 - x + 1$



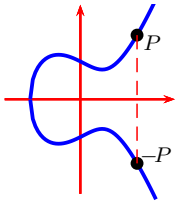
$y^2 = x^3 + x$

So long as the discriminant is not zero it can be shown that any elliptic curve given by $y^2 = x^3 + ax + b$ can be used to define an abelian group $E(a, b)$ consisting of all points (x, y) on the curve together with another element denoted by O. In this group the binary operation $+$ is defined for two points P and Q depending on four possible cases:

1. If $P = O$ or $Q = O$ then $P + Q$ is just taken to be P (if $Q = O$) or else Q (if $P = O$), so that O acts as the identity for the group.

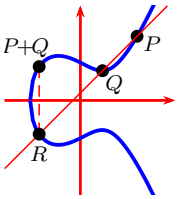
2. If $P = (x, y)$ and $Q = (x, -y)$, so that Q is the mirror image of P in the x-axis then $P + Q$ is taken to be O, so that $Q = -P$ is the inverse of P in the group. Algebraically:

$$-(x, y) = (x, -y)\,.$$



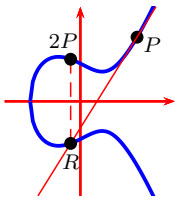
3. If $P = (x, y)$ and $Q = (x', y')$ with $x \neq x'$ then $P + Q$ is taken to be $-R$ where R is the point where the line through P and Q meets the elliptic curve a third time. Algebraically, the slope of the line through P and Q is given by $m = \frac{y'-y}{x'-x}$ and the following equation can be derived:

$$(x, y) + (x', y') = \left(m^2 - x - x', 2mx + mx' - m^3 - y\right).$$



4. If $P = Q$ then $P + Q = 2P$ is taken to be $-R$ where R is the point where the tangent line through P meets the elliptic curve again. Algebraically, the slope of the tangent is given by $m = \frac{3x^2+a}{2y}$ and the following equation can be derived:

$$2\,(x, y) = \left(m^2 - 2x, 3mx - m^3 - y\right).$$

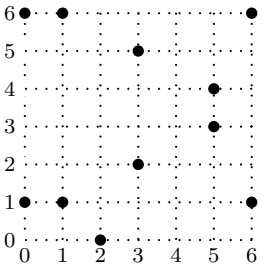


In Elliptic Curve Cryptography a finite field is used instead of $\mathbb{R}$. Typically if ECC is implemented in software then the field $\mathbb{Z}_p$ where p is a prime is chosen and an elliptic curve $y^2 = x^3 + ax + b$ with $4a^3 + 27b^2 \neq 0$ is used (called a *prime curve*), and addition calculations are performed using the previous algebraic equations:

$$\begin{aligned}(x, y) + (x', y') &= \left(m^2 - x - x', 2mx + mx' - m^3 - y\right)\\ 2\,(x, y) &= \left(m^2 - 2x, 3mx - m^3 - y\right).\end{aligned}$$

The number of elements in the resulting abelian group can be shown to be somewhere between $p + 1 - 2\sqrt{p}$ and $p + 1 + 2\sqrt{p}$. If instead ECC is implemented in hardware (where rapid bitwise operations are advantageous) then the field $GF(2^m)$ where $m \geq 1$ is chosen and an elliptic curve $y^2 + xy = x^3 + ax^2 + b$ with $b \neq 0$ is used (called a *binary curve*).

As an example, the elliptic curve $y^2 = x^3 - x + 1$ over the field $\mathbb{Z}_7$ has 11 solutions, the points $(0, 1)$, $(0, 6)$, $(1, 1)$, $(1, 6)$, $(2, 0)$, $(3, 2)$, $(3, 5)$, $(5, 3)$, $(5, 4)$, $(6, 1)$, $(6, 6)$. Including the identity O results in a group $E(-1, 1)$ with 12 elements. Note that for any point (x, y) in the abelian group its inverse is given by $-(x, y) = (x, -y) = (x, 7 - y)$ resulting in symmetry in the illustrated grid. So for example $-(5, 3) = (5, 4)$ and $-(2, 0) = (2, 0)$.



Taking $P = (5, 3)$ the equation for $2P$ gives that $m = \frac{3 \cdot 5^2 - 1}{2 \cdot 3} = 4 \cdot 6^{-1} = 3$ (using the fact that $6^{-1} = 6$ in the field $\mathbb{Z}_7$) and so:

$$2P = \left(3^2 - 2 \cdot 5, 3 \cdot 3 \cdot 5 - 3^3 - 3\right) = (6, 1).$$

Larger multiples nP of a point P can be found by repeated doubling of the previous value, obtaining $P, 2P, 4P, 8P, 16P, \ldots$ in $\Theta(\log_2 n)$. For some particular curves and values of p there are more efficient algorithms (such as the NIST P192 curve defined using the prime $p = 2^{192} - 2^{64} + 1$ where nP can be found using up to 38 addition and 192 doubling operations, quite feasible for a limited device). However, for a cryptanalyst to determine n given P and nP by brute force would require checking $P, 2P, 3P, 4P, 5P, \ldots$ in $\Theta(n)$, which is infeasible for large values of n (a brute force attack on the NIST P192 curve would require on average approximately $\frac{1}{2}p \approx 3 \times 10^{57}$ additions, completely infeasible on any computer). Hence the security of ECC relies on the difficulty of computing the integer n given the points P and nP, which is named the *elliptic curve discrete logarithm problem*.

ECC can be utilized for key exchange by making public the chosen field (p or 2^m), the curve (a and b values in the field), and a point G in the resulting abelian group for which there are many multiples $G, 2G, 3G, \ldots, nG$ that are all distinct (eventually the multiples start to repeat when for some integer value of n called the *order* of G one finds that $nG = O$). When two processes want to exchange a secret key each selects its own secure random number $s < n$ as its private key and calculates a public key $Q = sG$, which is communicated to the other process. When each process receives the public key $Q' = s'G$ of the other process (but not knowing the other private key s'), it calculates the secret key K:

$$K = sQ'.$$

Note that as in Diffie-Hellman key exchange this gives the same key for both processes as:

$$s'Q = \underbrace{Q + Q + \cdots + Q}_{s' \text{ times}} = \underbrace{G + G + \cdots + G}_{s \cdot s' \text{ times}} = \underbrace{Q' + Q' + \cdots + Q'}_{s \text{ times}} = sQ'.$$

JCA provides support for Elliptic Curve Cryptography, including the interface `ECField` with implementing classes `ECFieldFp` and `ECFieldF2m` for representing finite fields of order p and 2^m respectively, the interface `ECKey` with subinterfaces `ECPublicKey`, `ECPrivateKey` and implementing classes `ECPublicKeySpec`, `ECPrivateKeySpec`, and classes `EllipticCurve`, `ECParameterSpec`, `ECGenParameterSpec` for creating elliptic curves and suitable parameters for the cipher. The cipher for ECC is named `ECIES` and the key agreement is named `ECDH`. For example, the Eclipse project `AndroidECKeyExchangeDemo` has the activity `AndroidECKeyExchangeActivity` that demonstrates ECC key exchange using the fixed elliptic curve secp256r1.

The Bouncy Castle lightweight API also supports Elliptic Curve Cryptography, including the class `ECCurve` with subclasses `ECCurve.Fp` and `ECCurve.F2m` for representing prime and binary curves, `ECPoint` for representing a point on an elliptic curve, and `ECDomainParameters` for suitable parameters for the cipher. The MIDlet `ECKeyExchangeMIDlet` demonstrates how ECC can be used for key exchange. It creates an `ECCurve` for the fixed elliptic curve secp256r1. Using the parameters for the curve and a point (`GX`,`GY`) on the curve an `AsymmetricCipherKeyPair` is generated and its public key made available to the other process. Once the public key is received from the other process a `BasicAgreement` is used to calculate the secret key. Note that the generation of the key pair is very computationally intensive, in practice this step would need to be performed in a background thread.

Besides the shorter key size required for ECC as compared to other public-key encryption systems, ECC encryption is not vulnerable to *index calculus* attacks that are used against RSA and Diffie-Hellman, which make use of certain special properties of the groups created by RSA and Diffie-Hellman, properties which are not found in the ECC groups $E(a, b)$. The best known current attack against ECC is believed to be the *Pollard ρ attack*, one of a class of cryptanalysis techniques known as a *collision search*. However, even its computational requirements grow exponentially with the size of the field, quickly becoming infeasible for moderate values of p.

Exercise 2.4 (ECC Key Exchange) *Use either the Android example* `AndroidECKeyExchangeActivity` *or the Java ME Bouncy Castle lightweight example* `ECKeyExchangeMIDlet` *to prepare a mobile application that uses SMS communication to perform an ECC key exchange.*

ECC KEY EXCHANGE

```
/**
* Android Activity that demonstrates ECC key exchange using the fixed elliptic curve
    secp256r1
* @author Andrew Ensor
*/
package aut.hss;
...
import java.math.BigInteger;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.ECField;
import java.security.spec.ECFieldFp;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECPoint;
import java.security.spec.EllipticCurve;
import java.util.Arrays;
import javax.crypto.KeyAgreement;
...
public class AndroidECKeyExchangeActivity extends Activity implements
OnClickListener
{
private Button exchangeButton;
private TextView statusTextView;
// well-known elliptic curve secp256r1 (256 bit prime, 32 byte)
private static final BigInteger P = new BigInteger
("FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFF...", 16);
private static final BigInteger A = new BigInteger
("FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFF...", 16);
private static final BigInteger B = new BigInteger
("5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3...", 16);
private static final BigInteger GX = new BigInteger
("6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13...", 16);
private static final BigInteger GY = new BigInteger
("4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB64...", 16);
private static final BigInteger N = new BigInteger
("FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9C...", 16);
private static final int H = 1; // cofactor

// inner class that represents one end of a key exchange
private static class KeyExchangeProcess
{
private ECPublicKey publicKey;
```

```
private ECPrivateKey privateKey;
\end{verbatim}\hfill \emph{cont-}\end{program}\end{figure*}%
\begin{figure*}\begin{program}\emph{-cont}\begin{verbatim}
public KeyExchangeProcess()
{ // create the elliptic curve
ECField field = new ECFieldFp(P);
EllipticCurve curve = new EllipticCurve(field, A, B);
// use existing precalculated values of G and N
ECPoint G = new ECPoint(GX, GY);
ECParameterSpec ecSpec = new ECParameterSpec(curve, G, N, H);
// create public and private keys
try
{ KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
kpg.initialize(ecSpec); // size of key in bits
System.out.println("About to generate a key pair");
KeyPair keyPair = kpg.generateKeyPair();
System.out.println("Finished generating a key pair");
privateKey = (ECPrivateKey)keyPair.getPrivate();
publicKey = (ECPublicKey)keyPair.getPublic();

}
catch (NoSuchAlgorithmException e)
{ System.err.println
("Encryption algorithm not available: " + e);
}
catch (InvalidAlgorithmParameterException e)
{ System.err.println
("Invalid Parameter in Encryption algorithm: " + e);
}
}

public ECPublicKey getPublicKey()
{ return publicKey;
}

public byte[] calculateSecretKey(ECPublicKey otherPublicKey)
{ // generate secret key using Elliptic Curve Diffie-Hellman
byte[] key = null;
try
{ KeyAgreement ka = KeyAgreement.getInstance("ECDH");
ka.init(privateKey); // initialize with own private key
ka.doPhase(otherPublicKey, true);
key = ka.generateSecret();
}
catch (NoSuchAlgorithmException e)
{ System.err.println
("Key agreement algorithm not available: " + e);
}
catch (InvalidKeyException e)
{ System.err.println
("Invalid key in key agreement algorithm: " + e);
}
return key;
}
}
}
```

ECC key exchange via the Bouncy Castle

```
/**
   A MIDlet that demonstrates ECC key exchange via the Bouncy Castle
   lightweight API
   @author Andrew Ensor
```

```
*/
...
public class ECKeyExchangeMIDlet extends MIDlet
   implements CommandListener
{
   private Display display;
   private Form form;
   private Command exitCommand, exchangeCommand;
   // well-known elliptic curve secp256r1 (256 bit prime, 32 byte)
   private static final BigInteger P = new BigInteger
      ("FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFF...", 16);
   private static final BigInteger A = new BigInteger
      ("FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFF...", 16);
   private static final BigInteger B = new BigInteger
      ("5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3...", 16);
   private static final BigInteger GX = new BigInteger
      ("6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13...", 16);
   private static final BigInteger GY = new BigInteger
      ("4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB64...", 16);
   private static final BigInteger N = new BigInteger
      ("FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9C...", 16);
   private static final int H = 1; // cofactor

   ...
   // implementation of the CommandListener interface
   public void commandAction(Command c, Displayable s)
   { if (c == exitCommand)
         exitMIDlet();
      else if (c == exchangeCommand)
      { KeyExchangeProcess processA = new KeyExchangeProcess();
         KeyExchangeProcess processB = new KeyExchangeProcess();
         CipherParameters publicKeyA = processA.getPublicKey();
         CipherParameters publicKeyB = processB.getPublicKey();
         BigInteger secretKeyA=processA.calculateSecretKey(publicKeyB);
         BigInteger secretKeyB=processB.calculateSecretKey(publicKeyA);
         if (secretKeyA.equals(secretKeyB))
         { form.append("Secret keys do agree and have length "
               + (secretKeyA.bitLength()+7)/8 + " bytes\n");
            form.append("Secret key: ".concat(secretKeyA.toString()));
         }
         else
         { form.append("Error: secret keys do not agree!");
            form.append("Secret key: ".concat(secretKeyA.toString()));
            form.append("Secret key: ".concat(secretKeyB.toString()));
         }
      }
   }

   // inner class that represents one end of a key exchange
   private class KeyExchangeProcess
   {
      private CipherParameters publicKey, privateKey;

      public KeyExchangeProcess()
      { // create the elliptic curve
         ECCurve.Fp curve = new ECCurve.Fp(P, A, B);
         // use existing precalculated values of G and N
         ECPoint G = new ECPoint.Fp(curve,
            new ECFieldElement.Fp(P,GX), new ECFieldElement.Fp(P,GY));
         ECDomainParameters params = new ECDomainParameters(curve, G,
            N);
         // create public and private keys
         SecureRandom generator = new SecureRandom();
```

```
        ECKeyGenerationParameters ecSpec
           = new ECKeyGenerationParameters(params, generator);
        ECKeyPairGenerator kpg = new ECKeyPairGenerator();
        kpg.init(ecSpec);
        System.out.println("About to generate a key pair");
        AsymmetricCipherKeyPair keyPair=kpg.generateKeyPair(); //slow
        System.out.println("Finished generating a key pair");
        publicKey = keyPair.getPublic();
        privateKey = keyPair.getPrivate();
     }

     public CipherParameters getPublicKey()
     { return publicKey;
     }

     public BigInteger calculateSecretKey
        (CipherParameters otherPublicKey)
     { // generate secret key using
        BasicAgreement ba = new ECDHBasicAgreement();
        ba.init(privateKey); // initialize with own private key
        BigInteger key = ba.calculateAgreement(otherPublicKey);
        return key;
     }
  }
}
```

Chapter 3

Authentication

3.1 Message Authentication

Symmetric and public-key encryption both provide confidentiality of messages, ensuring that only the intended recipient can decrypt the ciphertext and obtain the original plaintext. However, unless the recipient knows the format of the plaintext to expect, such as English text, a recognizable image, or the response of a nonce value, the recipient can not be sure of the actual source of a particular message, or whether it may have been tampered with en route. The difficulty stems from the inability of a recipient to determine what is valid plaintext from a block that was encrypted with an incorrect key. Some file formats start with a recognizable header, but this might not inhibit an adversary replacing later blocks of ciphertext with their own blocks in an attempt to disrupt the system (in fact it might be better to put recognizable information at the end of the file rather than at the start if CBC mode encryption were used).

The verification of the integrity of a message is known as *message authentication*, ensuring that the message did originate from the alleged source and has not been altered in any way. This is achieved by adding extra information to the message calculated from its contents. One simple approach is to append an error-detecting code known as a *frame check sequence* (FCS) or *checksum* to the message before the message is encrypted, either with a secret key for a symmetric cipher or with the private key of a public-key cipher of the sender. When the recipient decrypts the message the FCS can be recalculated and compared with the value sent with the message. If the two FCS values agree then the recipient presumes the message was not altered between when the two values were calculated. So long as it is certain that the key used for encryption was kept confidential it can be sure that the message was encrypted by the alleged source.

More generally, a *(cryptographic) hash code* or *message digest* is a fixed-length value computed for a message which acts as a fingerprint for it. Ideally, a function h used to calculate hash codes should be able to be applied to messages of arbitrary lengths and be relatively quick to compute. For security, a hash function is designed to be *one-way*, meaning that given a hash code y it is computationally infeasible to create a message M which has the hash code y ($h(M) = y$). Hash functions do not need to be one-to-one functions, so it is possible that there are two messages M and M' which have the same hash code ($h(M') = h(M)$), in which case a *collision* is said to occur. They should however have *weak collision resistance*, meaning that given a message M it is computationally infeasible to create another message M' which has the same hash code ($h(M') = h(M)$).

Once a suitable hash function has been devised, it can be used for the authentication of a message. If the message itself should remain confidential then the hash code can be appended to the plaintext before it all gets encrypted as a single message. Then when the intended recipient decrypts the message it can check whether the appended hash code matches that of the decrypted message, and so determine whether the message was authentic.

Often however, only authentication of a message is required, as the message itself might need to be readable by all processes in the system (such as a broadcast) or else confidentiality might

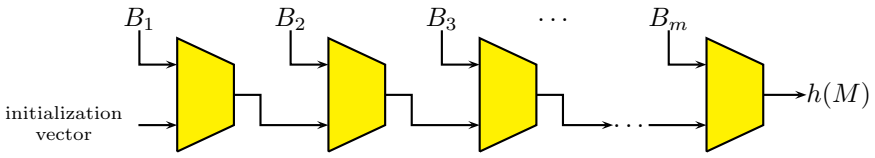
already be provided by the communication protocol. In this case just the hash code itself needs to be encrypted with the private key of the sender, and the encrypted hash code appended to the plaintext message. As only the holder of the key can produce the valid encrypted hash code, any recipient can decrypt the hash code to determine whether the message M was actually created by the holder of the private key. It is important to note that if an adversary could devise a message M' with exactly the same hash code as the original message M, then M could be substituted by M' and appended with the original encrypted hash code without the encryption ever needing to be broken, resulting in a forged message. Thus it is essential for authenticity that the hash function have weak collision resistance.

One simple hash code is the *longitudinal redundancy check*, which calculates an n-bit hash code by breaking a message M into blocks $B_1, B_2, \ldots, B_m$ each of length n, padding with zeros if required and taking a bitwise XOR of each block:

$$h(M) = B_1 \oplus B_2 \oplus \cdots \oplus B_m.$$

Often there are regularities in the blocks, such as for ASCII strings where the most-significant bit of each byte is typically 0. To avoid such regularities reducing the possible hash values the hash code might be rotated one bit before each XOR is calculated. However, the longitudinal redundancy check does not have weak collision resistance, given a message M it is not difficult to modify it slightly into a message M' (perhaps changing the amount for an on-line payment or a name) while still maintaining the same hash code. Even if the hash code were encrypted with the plaintext message using a secure symmetric block cipher with CBC mode this particular hash function might not be able to detect a reordering of the ciphertext blocks by an adversary.

Most contemporary hash functions make use of multiple rounds in which the message is used block by block $B_1, B_2, \ldots, B_m$ to calculate its hash code. In each round the result from the



previous round is used along with the next block of the message as input to a *compression function* which is designed to produce as few collisions as possible, with a fixed initialization vector used for the first round. So long as the compression function is collision resistent it can be shown that the resulting hash function will also be collision resistent. Thus attention can be focused on designing a good compression function for which it would be very difficult for an adversary to create a block that collides with another.

However, weak collision resistance might not be sufficient to ensure the authenticity of a message if an adversary is permitted to submit its messages to a process responsible for calculating and encrypting hash codes before they are sent. In such a situation the hash function must have *strong collision resistance*, meaning that it is computationally infeasible to create two messages M and M' that have the same hash code ($h(M') = h(M)$). Without strong collision resistance, an adversary could submit an innocuous message M to have its hash code calculated and encrypted by a process which is trusted in the system. Then the adversary could swap the message M with a non-approved message M' and append the correct encrypted hash code. When a recipient decrypted the hash code appended to M' it would presume that the trusted process did approve it, since that process did appear to have encrypted a valid hash code for it.

This form of attack is known as a *birthday attack*, due to a result from probability which states that the probability that in a room of n people there are at least two that have the same birthday is $1 - \frac{365!}{(365-n)!365^n}$. Surprisingly, with only $n = 23$ people the probability is over 0.5. The attack proceeds as follows. First two messages are prepared by the adversary, one innocuous message M such as:

```
Hi Jack, the paper Highly Secure Systems is great
```

and another subversive message M':

```
Please transfer all the funds from the account of Jack
```

Then many slight variants of each is produced, such as the following $2^5 = 32$ variants of M:

$$\begin{smallmatrix}\texttt{Hi}\\\texttt{Greetings}\end{smallmatrix}\ \texttt{Jack,}\ \begin{smallmatrix}\texttt{the}\\\texttt{my}\end{smallmatrix}\ \begin{smallmatrix}\texttt{paper}\\\texttt{course}\end{smallmatrix}\ \texttt{Highly Secure}\ \begin{smallmatrix}\texttt{Systems}\\\texttt{Software}\end{smallmatrix}\ \texttt{is}\ \begin{smallmatrix}\texttt{great}\\\texttt{fun}\end{smallmatrix}$$

and the following $2^5 = 32$ variants of M':

$$\begin{smallmatrix}\texttt{Please}\\\texttt{Immediately}\end{smallmatrix}\ \begin{smallmatrix}\texttt{transfer}\\\texttt{remove}\end{smallmatrix}\ \texttt{all}\ \begin{smallmatrix}\texttt{the}\\\texttt{available}\end{smallmatrix}\ \begin{smallmatrix}\texttt{funds}\\\texttt{money}\end{smallmatrix}\ \texttt{from}\ \begin{smallmatrix}\texttt{the}\\\ \end{smallmatrix}\ \texttt{account of Jack}$$

These variants could be automated by the adversary using a thesaurus, or more easily by alternating sequences of space-backspace-space with space-space-backspace. For each variant of M and of M' its hash code is calculated, and more variants produced until a collision is found between a variant of M and a variant of M'.

If variants $M, M_1, M_2, \ldots, M_{k-1}$ are produced with distinct hash codes, and variants $M', M'_1, M'_2, \ldots, M'_{k-1}$ are produced with distinct hash codes, then the probability that any selected M_i has the same hash code as any particular M'_j is $1/N$ (presuming all hash codes are equally likely), where N is the number of possible values of the hash code. Thus the probability that M_i does not have the same hash code as any of the k variants of M' is $\left(1 - \frac{1}{N}\right)^k$. If N is large this probability is very close to 1, making a brute-force attack infeasible. However, there are also k variants of M, so the probability that none of its variants have the same hash code as any of the variants of M' is $\left(\left(1 - \frac{1}{N}\right)^k\right)^k = \left(1 - \frac{1}{N}\right)^{k^2}$. This probability drops below 0.5 when:

$$k > \sqrt{\frac{\ln 0.5}{\ln(1 - 1/N)}} \approx 0.83\sqrt{N}.$$

If the hash function produces hash codes with m bits, so $N = 2^m$, then producing $k = 2^{m/2}$ variants of M and M' will probably be sufficient for a successful birthday attack. This makes any hash function which produces hash codes with less than about 128 bits vulnerable.

A *message authentication code* (MAC) is a type of hash code, but where the hash function also takes a secret key as input as well as a message of arbitrary length. Hence only the holders of the secret key are able to recalculate a MAC and verify the authenticity of a message. Essentially, a MAC acts like hash code encrypted using a secret key rather than a private key. Unlike a symmetric cipher, a MAC hash function need not be one-to-one which can make it harder for a cryptanalysis attack on the key used by a MAC. The inclusion of the secret key in the hash function also increases the difficulty of attacks. So long as only the receiver and the sender of a message share the secret key used in calculating the MAC, the receiver can be assured the message was not altered and did originate from the sender (however, it can not prove to others that the sender and not itself was the creator of the MAC).

The *Data Authentication Code* (DAC) was a popular MAC based on the DES cipher. Essentially, it processes a message in 64-bit blocks by encrypting each block using CBC mode with a zero initialization vector, using padding with zeros if required in the final block. Either the entire final encrypted block is used as the DAC, or else just some of the left-most bits of that block. Despite the popularity of DAC, due to security weaknesses in DES the DAC is being replaced by newer algorithms that are more resistant to attack.

Exercise 3.1 (Implementation of DAC) *Prepare an implementation of the Data Authentication Code using a DES cipher.*

3.2 Hash and MAC Algorithms

The *Message Digest 5* (MD5) algorithm has been a popular cryptographic hash function that produces a 128-bit hash code for a message. It processes a variable length message by breaking it into 512-bit blocks and using the hash code produced from one round as the initialization vector for the next round, where the first round uses the fixed IV `67452301EFCDAB8998BADCFE10325476`. The message is padded so that its length is divisible by 512 by first appending a single 1 bit, followed by as many 0 bits as required to leave exactly 64 bits available in the final block which are used to store the number of bits in the message (hence MD5 is only designed for a message up to 2^{64} bits in length).

MD5-ROUND(B, IV)

1 ▷ perform message digest MD5 on 512-bit block B using init vector IV
2 ▷ create 2D array R of left rotate amounts for each iteration of loop
3 $R \leftarrow \{\{7, 12, 17, 22\}, \{5, 9, 14, 20\}, \{4, 11, 16, 23\}, \{6, 10, 15, 21\}\}$
4 ▷ create integer array T of constants using sin function in radians
5 **for** $i \leftarrow 0$ **to** 63 **do**
6 $T[i] \leftarrow \lfloor abs(sin(i + 1)) \cdot 2^{32} \rfloor$
7 ▷ split block B into 16 words $W[0], \ldots, W[15]$ each 32-bit integer
8 **for** $k \leftarrow 0$ **to** 15 **do**
9 $W[k] \leftarrow B[32k \ldots 32k + 31]$
10 ▷ create 32-bit int buffer values a, b, c, d from initialization vector
11 $a \leftarrow IV[0 \ldots 31]$
12 $b \leftarrow IV[32 \ldots 63]$
13 $c \leftarrow IV[64 \ldots 95]$
14 $d \leftarrow IV[96 \ldots 127]$
15 **for** $i \leftarrow 0$ **to** 63 **do**
16 **if** $0 \leq i \leq 15$ **then**
17 $j \leftarrow (b \wedge c) \vee (\overline{b} \wedge d)$
18 $k \leftarrow i$
19 **else if** $16 \leq i \leq 31$ **then**
20 $j \leftarrow (d \wedge b) \vee (\overline{d} \wedge c)$
21 $k \leftarrow (5i + 1) \bmod 16$
22 **else if** $32 \leq i \leq 47$ **then**
23 $j \leftarrow b \oplus c \oplus d$
24 $k \leftarrow (3i + 5) \bmod 16$
25 **else** ▷ $48 \leq i \leq 63$
26 $j \leftarrow c \oplus (b \vee \overline{d})$
27 $k \leftarrow 7i \bmod 16$
28 $r \leftarrow R[i/16][i \bmod 4]$ ▷ number of bits by which to rotate left
29 $temp \leftarrow d$
30 $d \leftarrow c$
31 $c \leftarrow b$
32 $b \leftarrow ((leftRotate(a + j + T[i] + W[k], r) + b) \bmod 2^{32}$
33 $a \leftarrow temp$
34 ▷ calculate the message digest from a, b, c, d
35 $a \leftarrow a + IV[0 \ldots 31]$
36 $b \leftarrow b + IV[32 \ldots 63]$
37 $c \leftarrow c + IV[64 \ldots 95]$
38 $d \leftarrow d + IV[96 \ldots 127]$
39 **return** 128-bit value $abcd$ as the message digest

MD5-PADDING(M)

1 ▷ pad the message M including appending the 64-bit integer length of M
2 $l \leftarrow (length(M) + 65) \bmod 512$ ▷ number of occupied bits in last block
3 append bit 1 to M
4 **for** $i \leftarrow l + 1$ **to** 512 **do**
5 append bit 0
6 append $length(M)$ as a 64-bit integer value

Each 512-bit block is split into 16 integer 32-bit values which are used to modify four 32-bit buffer values a, b, c, d created from the initialization vector. These buffer values pass through 64 iterations of a loop resulting in a hash value for the block (used as the initialization vector for the next round) as can be seen by the MD5-ROUND algorithm.

Since 1996 various security vulnerabilities have been found in the MD5 algorithm, including an attack that can find a collision in less than a minute. Hence, MD5 is very vulnerable to a birthday attack, and no longer considered sufficiently secure to provide authentication. For example, the 128-byte message:

```
D1  31  DD  02  C5  E6  EE  C4  69  3D  9A  06  98  AF  F9  5C
2F  CA  B5  87  12  46  7E  AB  40  04  58  3E  B8  FB  7F  89
55  AD  34  06  09  F4  B3  02  83  E4  88  83  25  71  41  5A
08  51  25  E8  F7  CD  C9  9F  D9  1D  BD  F2  80  37  3C  5B
D8  82  3E  31  56  34  8F  5B  AE  6D  AC  D4  36  C9  19  C6
DD  53  E2  B4  87  DA  03  FD  02  39  63  06  D2  48  CD  A0
E9  9F  33  42  0F  57  7E  E8  CE  54  B6  70  80  A8  0D  1E
C6  98  21  BC  B6  A8  83  93  96  F9  65  2B  6F  F7  2A  70
```

has the MD5 hash code `79054025255FB1A26E4BC422AEF54EB4`. But changing just the most-significant bit of only six bytes in this message results in a different message with the identical hash code. Despite its deficiencies, MD5 is still used to perform (non-secure) error checking on messages and as a component in more secure algorithms.

The *Secure Hash Algorithm* (SHA-1) has been a popular successor to MD5. It was developed by the US National Security Agency based on a predecessor (MD4) of the MD5 algorithm, but produces a (more secure) 160-bit hash code using five 32-bit buffer values a, b, c, d, e and 80 iterations of a loop. It is employed in many popular security protocols such as SSL/TLS and used for the secure e-mail protocols Secure/Multipurpose Internet Mail Extensions (S/MIME) and Pretty Good Privacy (PGP). The `MessageDigest` class from the `java.security` package can be used in Java to calculate hash codes for hash functions such as `MD5` and `SHA-1`:

```
MessageDigest digest = MessageDigest.getInstance("SHA-1");
byte[] message = ...;
digest.update(message); // can update several times
byte[] hashCode = digest.digest(); // digest gets reset
```

In C# the class `SHA1Managed` from the `System.Security.Cryptography` namespace is instead used:

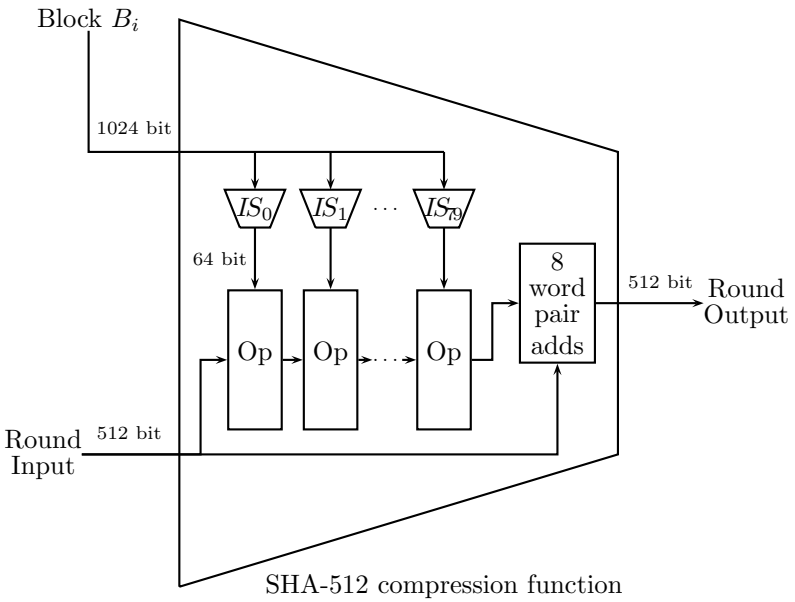
```
SHA1 digest = new SHA1Managed();
byte[] message = ...;
byte[] hashCode = digest.ComputeHash(message);
```

Although still widely used, the security of SHA-1 was considered suspect even before 2005 when an attack was found that produced collisions in SHA-1. The attack took advantage of the structure and some security weaknesses in SHA-1, requiring only 2^{69} operations to find a collision, 2048 times fewer than expected for a birthday attack. Hence, both MD5 and SHA-1 are being phased out in favour of more secure hash functions.

In response to the potential vulnerabilities of SHA-1 it has been modified to give three new algorithms, SHA-256, SHA-384, SHA-512, which give hash codes of length 256, 384, 512 bits respectively. SHA-512 processes a message using blocks of 1024 bits at a time, and first pads the message to 1024-bit blocks by appending a 1 bit followed by 0 bits and then a 128-bit integer giving the length of the message in bits. It uses eight 64-bit buffer values a, b, c, d, e, f, g, h and uses 80 iterations of a loop. In each iteration 64 bits are extracted from the 1024-bit block and passed as an *input sequence* into an operation along with the 512 bits of buffer values. After 80 iterations each of the eight updated buffer values is added to its original value (modulo 2^{64}) to produce a 512-bit hash code.

The *Whirlpool* algorithm uses a different approach to generating a hash function. It is based on the AES block cipher, but has been adapted to provide performance comparable to SHA-512, and processes a variable length message by breaking it into 512-bit blocks to produce a 512-bit hash code (128 bits was considered too susceptible to a birthday attack to be used by a hash function). It too starts by padding the message with a 1 bit followed by some 0 bits and a 256-bit integer length of the message, but uses 10 iterations of an S-box substitution, a rotation of the columns, a mix of rows, and an XOR with a round key which is obtained by performing the same iterations with a fixed initialization vector. Although it is now an ISO standard, and along with SHA-256, SHA-384, SHA-512 is one of the only hash functions endorsed by the New European Schemes for Signatures, Integrity, and Encryption (NESSIE), the default `SunJCE` security provider does not (yet) provide support for Whirlpool. However, support is available through other Java providers such as the Bouncy Castle provider using the algorithm name `WhirlpoolDigest`, and in the Lightweight API with the following code fragment:

```
Digest digest = new WhirlpoolDigest();
byte[] message = ...;
digest.update(message, 0, message.length);
```



SHA-512 compression function

```
byte[] hashCode = new byte[digest.getDigestSize()];
digest.doFinal(hashCode, 0);
```

Usually passwords are not stored as plaintext in a system (even on a process that might be considered relatively secure), but rather only ever stored in some encrypted form. This way, if a password file is obtained by an adversary only the encrypted keys are obtained. When a password needs to be checked by the system for validity, its plaintext form is encrypted (and so can be safely communicated) and compared with the stored encrypted key. Since the original password never need be obtained from the encrypted key, hash functions are usually chosen for the encryption as they are designed to be one-way (whereas ciphers are designed to be reversible). Although MD5 does not have strong collision resistance, it still provides some security against a *preimage attack*, where a cryptanalyst attempts to find a message that has a specific hash code, and so is still widely used for encrypting passwords.

Password based encryption (PBE) derives an secret key from a password. In order to make attempts to deduce the password more time consuming for an attacker, most PBE implementations mix a (fixed) random number called a *salt* with the key. Even if an attacker did manage to find the secret key corresponding to a password, the salt could be changed, thus totally changing the key (and if different salts were used around a system, each would have to be attacked separately). The following Java code fragment uses MD5 and DES to generate a PBE encrypted key from a password (specified as a `char[]` array since `String` objects are immutable), a salt, and an `int` that specifies the number of iterations to use in the PBE calculation:

```
char[] password = ...; // never stored as a String
byte[] salt = ...; // fixed random salt bytes
PBEKeySpec pbeSpec = new PBEKeySpec(password, salt, 20);
SecretKeyFactory keyFactory
   = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey key = keyFactory.generateSecret(pbeSpec);
```

The resulting secret key could then be used as an encrypted version of the password, or instead as the secret key for DES encryption:

```
Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
```

PBE works by first appending the salt to the password and then applying the hash function repeatedly to the result the specified number of iterations. In the case of an MD5 hash function, a 128 bit hash code is produced, which for a DES cipher the first 64 bits get used as the secret key and the remaining 64 bits as the initialization vector for CBC mode.

The Data Authentication Code described in Section 3.1 uses the DES algorithm with CBC mode to produce a MAC for a message. However, cryptographic hash functions such as MD5,

SHA-1 and Whirlpool execute faster than a cipher such as DES, so it is generally preferable to create a MAC from a hash function than from a symmetric cipher. The *Key-Hash Message Authentication Code* (HMAC) algorithm provides a way to generate a MAC by using a cryptographic hash function h combined with a secret key K. Any hash function which operates block by block on a message can be used, and HMAC is designed so that alternative hash functions can be easily plugged in, enabling hash functions to be replaced when more secure ones are required. HMAC produces a MAC with the same length as the hash function that it uses, so `HmacMD5` produces a 128-bit MAC and `HmacSHA512` produces a 512-bit MAC. It starts by checking whether the length of the key is exactly one block. If the key is less than one block it is padded with 0 bits on the left, whereas if it is more than one block $h(K)$ is calculated. This results in a key K^+ with length one block. Next, K^+ is passed through a bitwise XOR with the block *ipad*=`3636...36` (in hexadecimal). Then the message M is appended to this block and the hash code calculated, giving:

$$h\left(K^+ \oplus ipad \,||\, M\right).$$

For efficiency with multiple messages, the hash code $h\left(K^+ \oplus ipad\right)$ can be precalculated and used as the initialization vector for the calculation of each $h(M)$. Separately, K^+ is passed through an bitwise XOR with the block *opad*=`5C5C...5C`. The result has $h\left(K^+ \oplus ipad \,||\, M\right)$ appended to it and the hash code calculated again to give the MAC:

$$h\left(K^+ \oplus opad \,||h\left(K^+ \oplus ipad \,||\, M\right)\right).$$

Again, for efficiency when dealing with multiple messages, $h\left(K^+ \oplus opad\right)$ can be precalculated and used as the initialization vector for the calculation of each $h\left(h\left(K^+ \oplus ipad \,||\, M\right)\right)$. The JCA supports MAC algorithms such as HMAC with the `Mac` class:

```
KeyGenerator kg = KeyGenerator.getInstance("HmacSHA512");
SecretKey key = kg.generateKey();
Mac mac = Mac.getInstance("HmacSHA512");
mac.init(key);
byte[] message = ...;
byte[] hashCode = mac.doFinal(message);
```

Exercise 3.2 (SMS Message Authentication) *Enhance Exercise 2.4 so a Whirlpool hash code is appended to the SMS message which the receiver can use to authenticate the message.*

3.3 Digital Signatures

A *digital signature* is an encrypted sequence of bytes appended to a message that can be used to verify the sender of the message, when the signature was created, and authenticate the contents of the message. The advent of public-key encryption made digital signatures feasible, enabling an identifier of the sender, a timestamp, and a hash code of the message to be together encrypted using the private key of the sender. Any process with access to the corresponding public key could then decrypt the digital signature, recalculate the hash code for the message and verify the authenticity of the message (both the authenticity of the sender and the integrity of the message). If confidentiality of the message were important, the message together with its digital signature could be encrypted before it were sent.

Digital signatures also provide *non-repudiation*, unless another message with the same hash code has been created by the receiver or an adversary, or else the sender has released its private key, it cannot deny having placed its signature on the message. Note that instead using symmetric encryption of a message appended with a hash code does allow the receiver to authenticate the message, but does not itself provide non-repudiation as the sender could claim that the receiver (who also holds the secret key) might have produced the message itself, or that it is a replay attack by an adversary.

To avoid disputes between the sender and receiver over the validity of the timestamp provided in a digital signature any time-critical messages (such as on-line instructions for buying or selling stocks) might be sent via a mutually-trusted *arbitrator*, which itself checks the signature of the sender and puts its own timestamp on the message before encrypting it itself for the receiver.

Any standard public-key encryption algorithm such as RSA or ECC combined with a cryptographic hash function such as SHA-1 can be used to create a digital signature, so long as recipients are aware of which encryption algorithm and hash function have been chosen and the correct public key is available to the recipients. Instead, the *Digital Signature Algorithm* (DSA) is a standardized algorithm designed specifically for signing and verifying digital signatures which makes use of a hash function such as SHA-1. First, global public key components p, q, g are created and made available in the system (which can be shared by all DSA users in the system) using the following steps:

1. a 160-bit prime number q is selected (called the *sub-prime*),
2. an L-bit prime number of the form $p = mq+1$ for some integer multiple m is found, where L is either 1024 (default for SHA-1) or else 3072 (for SHA-256), 7680 (for SHA-384), or 15360 (for SHA-512),
3. a value $n < p - 1$ for which $n^m \bmod p > 1$ is selected, and the *base* g is taken to be $g = n^m \bmod p$.

Then the signer creates a private key x by selecting a secure random number $x < q$, and calculates a public key $y = g^x \bmod p$. In order to sign a message M using a cryptographic hash function h the signer uses the following steps:

1. a nonce value $k < q$ is selected for the message and its multiplicative inverse k^{-1} in $\mathbb{Z}_q$ is calculated using the Euclidean algorithm,
2. the signature is the pair (r, s) where $r = \left(g^k \bmod p\right) \bmod q$ and $s = \left(k^{-1}\left(h(M) + xr\right)\right) \bmod q$.

The recipient of a message M can verify a signature (r, s) for a received message M by obtaining the global public elements p, q, g, the public key y, and then use the following steps:

1. if $r \geq q$ or $s \geq q$ the signature is rejected,
2. the multiplicative inverse s^{-1} of s in $\mathbb{Z}_q$ is calculated,
3. the values $u_1 = h(M)s^{-1} \bmod q$ and $u_2 = rs^{-1} \bmod q$ are both calculated,
4. the value $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$ is calculated,
5. the signature is accepted as valid if and only if $v = r$.

The fact that correct DSA signatures do get accepted as valid can be verified once Fermat's Theorem is applied to show that $g^q \bmod p = 1$. The security of DSA is based on the difficulty of calculating the discrete logarithm x given the values g, p, and $y = g^x \bmod p$.

An advantage to using DSA instead of a standard public-key encryption algorithm such as RSA is that the only computationally intensive calculations for signing a message are the calculations of r and k^{-1}. Both these calculations can be performed beforehand for several nonce values k so that messages can be quickly signed.

In Java the keys required for DSA can be generated by first obtaining a `KeyPairGenerator`:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
```

and initializing it to either generate new key components:

```
kpg.initialize(1024);
```

or else use existing values of p, q, g:

```
DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);
kpg.initialize(dsaSpec);
```

Then a key pair is generated:

```
KeyPair keyPair = kpg.generateKeyPair();
PrivateKey privateKey = keyPair.getPrivate();
PublicKey publicKey = keyPair.getPublic();
```

From the two keys the values of p, q, g, x, y can be obtained by typecasting them to `DSAPrivateKey` and `DSAPublicKey` respectively. Typically the receiver is already provided with the public key y and so a `KeyFactory` would be used instead of a `KeyPairGenerator` to create the `PublicKey`:

```
DSAPublicKeySpec dsaPublicSpec=new DSAPublicKeySpec(y,p,q,g);
KeyFactory kf = KeyFactory.getInstance("DSA");
PublicKey publicKey = kf.generatePublic(dsaPublicSpec);
```

To sign a message a `Signature` object is created specifying the hash function (such as SHA-1) to use and is initialized with the private key:

```
Signature signer = Signature.getInstance("SHA1withDSA");
signer.initSign(privateKey);
byte[] message = ...;
signer.update(message);
byte[] signature = signer.sign();
```

To verify a signature the `Signature` object is instead initialized with the public key:

```
Signature signer = Signature.getInstance("SHA1withDSA");
signer.initVerify(publicKey);
byte[] message = ...;
byte[] signature = ...;
signer.update(message);
boolean verifies = signer.verify(signature);
```

The *Elliptic Curve Digital Signature Algorithm* (ECDSA) is a variant of DSA which utilizes elliptic curves instead of using exponentiation, enabling smaller keys to be used for the production of signatures. First the chosen field (prime p), elliptic curve (a and b values), point G with order n are made available in the system. The signer creates a private key x by selecting a random number $x < n$, calculates a public key $Q = xG$, and then uses the following steps:

1. a nonce value $k < n$ is selected (relatively prime to n) for which the point $kG = (x, y)$ has x-coordinate with $x \bmod n \neq 0$ and for which $k^{-1}\left(h(M) + x^2\right) \bmod n \neq 0$,
2. the signature is (r, s) where $r = x \bmod n$ and $s = k^{-1}\left(h(M) + xr\right) \bmod n$.

The signature can then be validated by calculating $u_1 = h(M)s^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$, and checking whether the x-coordinate of $u_1G + u_2Q$ is r. ECDSA is supported by the Bouncy Castle Lightweight API.

The class `DSAFileSigner` demonstrates how DSA can be used to add a signature to a text file, and later verify that the signature is still valid. If between the signing and the verification the file is modified in some way the signature will no longer be correct (unless a collision is found). In this example the signature has been stored using a Base 64 encoding, which stores the signature as printable text characters rather than a `byte[]` array. For simplicity it has used the non-standard `sun.misc` classes `BASE64Encoder` and `BASE64Decoder` to perform the encoding and decoding between a `byte[]` array and a `String` of printable characters. These classes might not be supported in some Java virtual machines and so should not be used in a production environment.

Digital Signature Algorithm

```
/**
   A class that demonstrates how DSA can be used to add a digital signature to a text
       file, using line "SIGNATURE:" followed by base 64 encoding of signature, and
       verify if signature is still valid.
   @author Andrew Ensor
*/
...
public class DSAFileSigner
{
   public static String SIG_HEADER = "SIGNATURE:";
   private PrivateKey privateKey;
   private PublicKey publicKey;

   public DSAFileSigner()
   { try
      { // generate the public and private keys for DSA
         KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
```

```
      kpg.initialize(1024);
      KeyPair keyPair = kpg.generateKeyPair();
      privateKey = keyPair.getPrivate();
      publicKey = keyPair.getPublic();
   }
   catch (NoSuchAlgorithmException e)
   { System.err.println("Encryption algorithm not available: "+e);
   }
}

// returns a Base64 encoded DSA signature for chars obtained from reader and then
    closes the reader stream
public String sign(Reader reader) throws IOException
{ byte[] signature = null;
   BufferedReader br = new BufferedReader(reader);
   try
   { Signature signer = Signature.getInstance("SHA1withDSA");
      signer.initSign(privateKey);
      String line = br.readLine();
      while (line != null)
      { signer.update(line.getBytes());
         line = br.readLine();
      }
      signature = signer.sign();
   }
   ...
   finally
   { try
      { br.close();
      }
      catch (IOException e)
      {} // ignore
   }
   return (new BASE64Encoder()).encode(signature);
}

// obtains the information from reader until the signature header is found, and
    then tries to verify the signature
public boolean verify(Reader reader) throws IOException
{
   BufferedReader br = new BufferedReader(reader);
   try
   { Signature signer = Signature.getInstance("SHA1withDSA");
      signer.initVerify(publicKey);
      String line = br.readLine();
      while (line!=null && !line.equals(SIG_HEADER))
      { signer.update(line.getBytes());
         line = br.readLine();
      }
      if (line == null)
         System.out.println("No signature found");
      else
      { // obtain the signature from the remaining information
         StringBuffer sigString = new StringBuffer();
         line = br.readLine();
         while (line != null)
         { sigString.append(line);
            line = br.readLine();
         }
         byte[] signature = (new BASE64Decoder()).decodeBuffer
            (sigString.toString());
         return signer.verify(signature);
      }
```

```
    }
    catch (NoSuchAlgorithmException e)
    { System.err.println("Encryption algorithm not available: "+e);
    }
    catch (InvalidKeyException e)
    { System.err.println("Invalid key: "+e);
    }
    catch (SignatureException e)
    { System.err.println("Signature algorithm exception: "+e);
    }
    finally
    { try
      { br.close();
      }
      catch (IOException e)
      {} // ignore
    }
    return false;
  }
  ...
}
```

A *Base 64* encoding is a binary to text encoding scheme where an arbitrary number of bytes is converted into a sequence of printable characters, using the 64 printable characters `A,B,...,Z,a,b,...,z,0,1,...,9,+,/` (formats such as MIME and PGP use these characters but other encoding schemes might use different characters instead of `+` and `/`). Given a `byte[]` array to be encoded (such as a digital signature or other `byte` data), a 24-bit buffer is used to process the bits in the array. The bytes are processed in blocks of three bytes at a time and converted into four printable characters by splitting the buffer into four 6-bit parts and using a table of the corresponding character encodings for each 6-bit value. If the length of the array is not a multiple of three then either one or two `0` bytes is appended to make a full block and the encoded output ends with that number of `=` characters to indicate the number of padding bytes.

Base 64 Encoding

6-bit Value	*Character*
000000	A
000001	B
⋮	⋮
011001	Z
011010	a
011011	b
⋮	⋮
110011	z
110100	0
110101	1
⋮	⋮
111101	9
111110	+
111111	/

For example, the bytes `73,65,63,75,72,69,74,79` (the hexadecimal representation of the string `security`) would be encoded as follows:

73	65	63	75	72	69	74	79	00
01110011	01100101	01100011	01110101	01110010	01101001	01110100	01111001	00000000
c	2	V	j	d	X	J	p	d	H	k	=

resulting in the printable string `c2VjdXJpdHk=`. Formats such as MIME also insert a newline (a combination of carriage return and newline characters) every 76 encoded characters so that the resulting strings are not too long for e-mail systems.

Exercise 3.3 (Base 64 Encoding) *Prepare a program that performs Base 64 encoding and decoding.*

3.4 Certificates and Key Storage

A *(public-key) certificate* is a document that contains an identifier for a process and its public key, together with a digital signature of the information prepared by a signer that is trusted to ensure the authenticity of the identifier and the public key. The signer is often a *certificate authority* (CA), a well-known service that is entrusted to securely check information before signing and issuing a certificate. The CA prepares a digital signature using a public key that

has been securely distributed around the system as a *self-signed* or *root* certificate, a certificate containing the public key and signed using the corresponding private key. Certificates make public-key encryption practical in large distributed systems, where it is usually infeasible to securely distribute secret keys. A process can pass its public key to another process simply by sending a certificate that has been signed by a CA trusted by the other process. When a certificate is received the digital signature can be verified using the public key of the CA to check the authenticity of the identifier and public key.

The most commonly used standard format for public-key certificates is *X.509* which holds de-

Version: *1 (default), 2, or 3* Serial Number: *integer ID unique amongst certificates issued by that CA* Signature Algorithm: *identifier for algorithm used to sign certificate* Issuer: *X.500 name of certificate issuer that created and signed certificate*
Validity Not Before: *first date from which certificate is valid* Not After: *last date on which certificate is valid*
Subject: *name of subject for whom the certificate is issued*
Subject Public Key Info Public Key Algorithm: *identifier for algorithm used to create public key* Key: *public key for the subject*
Issuer Unique Identifier: *optional unique name for certificate issuer* Subject Unique Identifier: *optional unique name for certificate subject*
Extensions to X.509 certificate
Signature Algorithm: *identifier for algorithm used to sign certificate* Signature: *digital signature of other fields of certificate*

tails about the issuer of the certificate, the period of validity of the certificate, details about the subject for whom the certificate is issued including its public key, followed by a digital signature of the information signed using the issuer's private key. X.509 is particularly convenient for algorithms such as ECC and DSA whose keys have multiple components as all the components get included in a single certificate.

The `java.security.cert` package contains the class `X509Certificate` that represents an X.509 certificate. The `getEncoded` method of `PublicKey` can be used to obtain a single `byte[]` array for an X.509 public key specification holding all the components of the public key (but the specification is not itself an X.509 certificate as it only holds the public key information and is not signed):

```
byte[] publicKeyBytes = publicKey.getEncoded();//X.509 format
String publicKeyAlg = publicKey.getAlgorithm();
```

This `byte[]` array can later be used to reconstruct the public key if the algorithm that was used to create the key is known:

```
X509EncodedKeySpec publicKeySpec
   = new X509EncodedKeySpec(publicKeyBytes);
KeyFactory keyFactory = KeyFactory.getInstance(publicKeyAlg);
PublicKey publicKey=keyFactory.generatePublic(publicKeySpec);
```

If instead an X.509 certificate has been received via an `InputStream` then the `X509Certificate` can be reconstructed using a `CertificateFactory`, in which case the validity of the certificate can be verified if the public key of the signer is known:

```
InputStream is = ...; // byte stream of X.509 certificate
CertificateFactory certFactory
   = CertificateFactory.getInstance("X.509");
X509Certificate certificate
   = (X509Certificate)certFactory.generateCertificate(is);
PublicKey signerPublicKey=...;//securely obtained from signer
PublicKey certPublicKey = certificate.getPublicKey();
boolean notExpired = certificate.checkValidity();
boolean validSignature=certificate.verify(signerPublicKey);
```

It is usually not practical to insist that all certificates in a large system be signed and issued by a single CA. Hence it may occur that a process needs to verify a certificate signed by a CA, but the process only has the public key for another signer. In this case a *certificate chain* can be used, which is a sequence of certificates where each certificate in the chain can be verified using the public key that is the subject in the following certificate, where the last certificate in the chain is already trusted. For instance, if a process A has a trusted certificate for process B then it can use the public key of B in that certificate to verify any certificate that has been signed by B (using the corresponding private key of B). If process B signed a certificate for process C then when A received this certificate it could verify the signature and thus obtain the public key of C from the certificate. Thus A could then verify any certificates it received that were signed by C. In this way when a certificate is received by a process it can check whether there is a certificate chain that leads to a certificate that it trusts.

As the validity of each certificate in a certificate chain is essential a CA typically has a *certificate revocation list* (CRL), which it maintains with an up-to-date list of certificates that it no longer considers valid (such as if the private key were found to have been compromised, or the certificate details were found to have been falsified). To verify a certificate it is therefore important to check that it is not in the CRL of the signer. The procedures for creating, managing, storing, distributing, verifying, and revoking certificates using standardized certificate formats, certificate authorities, and certificate revocation lists are collectively known as *public-key infrastructure* (PKI).

A *keystore* is a password-protected file or database that stores keys and trusted certificates. A Java keystore can be conveniently managed using the command line `keytool` utility (found in the `bin` folder of the Java SDK). Every entry in a keystore is assigned a unique string *alias* to identify it, which is specified with the `-alias` option. A new keystore is automatically created if one with the specified name does not already exist when a new key pair is generated or a certificate is imported. The password for a keystore can be changed using the `-storepasswd` option (note in practice `keytool` should be left to prompt for the passwords rather than passwords provided directly via the option `-storepass` in a command-line shell, as the operating system might cache previous commands so leave the password exposed):

```
keytool -storepasswd -keystore mykeystore.jks
```

The entries in a keystore can be listed using the `-list` option:

```
keytool -list -keystore mykeystore.jks
```

which displays the number of entries currently in the key store and an MD5 signature for each. Also including the `-v` option displays the entries in a human-readable format, whereas including the `-rfc` option displays the entries using Base 64 encoding.

A new public/private key pair is created using the option `-genkeypair`, where the key pair generation algorithm to use is specified with the `-keyalg` option. Besides generating a public/private key pair, the private key is stored securely using a password and the public key is placed in a self-signed certificate, signed using the private key. For example, a command line statement such as:

```
keytool -genkeypair -alias mykeyalias -keyalg RSA
   -keystore mykeystore.jks
```

generates a new key pair for RSA encryption together with a self-signed certificate, where both the private key and the self-signed certificate are assigned the specified alias of the entry. If the certificate is to be deployed to a server such as `http://localhost` then the first and last name of the subject must exactly match `localhost` (or the option `-dname "cn=localhost"` be included), otherwise the certificate would be rejected during verification. Any key pair or signature algorithm that is supported by a registered Java cryptographic security provider can be used.

A certificate can be exported from a keystore to a binary certificate file using the `-export` option, specifying the alias of the entry and the name of the new certificate file:

```
keytool -export -alias mykeyalias -file mycertfile.cer
   -keystore mykeystore.jks
```

The contents of the certificate file can be displayed using the `-printcert` option:

```
keytool -printcert -file mycertfile.cer
```

Certificate files exported from another keystore that are trusted can also be imported into the keystore with the `-import` option:

```
keytool -import -alias mykeyalias -file mycertfile.cer
   -keystore mykeystore.jks
```

The `keytool` utility attempts to build a certificate chain for the new certificate from a self-signed certificate and already trusted certificates in the keystore. If it fails to establish a certificate chain, then the certificate information is displayed and the user prompted to verify it (by independently checking the signature with the alleged issuer of the certificate). Some self-signed certificates for several popular CA are available in the `cacerts` keystore included with the Java Runtime (found in the `jre\lib\security` folder and with the initial keystore password `"changeit"`).

A standardized *certificate authentication request* (CAR) for requesting a certificate to be signed and issued by a certificate authority can be prepared using the `-certReq` option:

```
keytool -certReq -alias mykeyalias -file mycertreq.csr
   -keystore mykeystore.jks
```

When the reply has been received from the CA the self-signed certificate in the keystore can be replaced by the certificate chain signed by the CA.

The `KeyStore` class can be used by a program to access a keystore independently of how it stores entries. Either the default type of keystore can be used or else a particular type can be specified (such as `"JKS"` for a Java Key Standard keystore as produced by `keytool`, or `"PKCS12"` for a keystore in the PKCS#12 format). A default type of keystore can be loaded with the following statements:

```
String keyStoreName = ...;
char[] storepw = ...; // don't store password in a String
FileInputStream fis = new FileInputStream(keyStoreName);
KeyStore keystore
   = KeyStore.getInstance(KeyStore.getDefaultType());
keystore.load(fis, storepw);
fis.close();
```

(passing a `null InputStream` creates an empty keystore). Once a keystore has been loaded an `Enumeration<String>` of its aliases can be obtained using the `aliases` method. Existing keys and certificates can be read using the alias for each entry:

```
String keyAlias = ...;
char[] keypw = ...; // don't store password in a String
Key key = keystore.getKey(keyAlias, keypw);
String certAlias = ...;
Certificate certificate = keystore.getCertificate(certAlias);
```

New entries can also be added to the keystore or existing entries modified:

```
Key key = ...;
Certificate[] chain = ...; // certificate chain if PrivateKey
keystore.setKeyEntry(keyAlias, key, keypw, chain);
Certificate certificate = ...;
keystore.setCertificateEntry(certificate, certAlias);
```

which is convenient for providing secure persistent storage of keys. If changes have been made they can be saved to the keystore:

```
FileOutputStream fos = new FileOutputStream(keyStoreName);
keystore.store(fos, storepw);
fos.close();
```

The *Java Certification Path API* consists of classes and interfaces in the package `java.security.cert` for verifying certificate chains. The `CertPath` class represents a certificate chain, ordered starting with the target certificate and ending with a certificate signed by a trusted CA, so that the signer of one certificate has its public key as the subject in the next certificate in the chain.

For example, the following code fragment obtains a certificate chain from a keystore and uses it to create a `CertPath` object:

```
String certAlias = ...;
Certificate[] chain=keystore.getCertificateChain(certAlias);
CertificateFactory cf
   = CertificateFactory.getInstance("X.509");
CertPath path = cf.generateCertPath(Arrays.asList(chain));
```

Once a `CertPath` has been created it can be validated by the PKI X.509 certification path validation algorithm, which throws a `CertPathValidatorException` if the validation fails:

```
PKIXParameters params = new PKIXParameters(keystore);
params.setRevocationEnabled(false); // no CRL
CertPathValidator validator = CertPathValidator.getInstance
   (CertPathValidator.getDefaultType());
PKIXCertPathValidatorResult vr=(PKIXCertPathValidatorResult)
   validator.validate(path, params); // exception if fails
X509Certificate trustedCert
   = vr.getTrustAnchor().getTrustedCert();
```

The Certification Path API also provides support for certificate revocation lists, including the *Online Certificate Status Protocol* (OCSP) for determining the current status of a certificate.

Exercise 3.4 (Distributing Certificates) *Modify* `DSAFileSigner` *from Section 3.3 so that it uses a DSA key pair that is generated by* `keytool` *and stored in a keystore. Then export the DSA public key certificate to another keystore which is used by another program to verify the signature of the file.*

Chapter 4

Network Security

4.1 Authentication Services

User authentication is vital to the security of a distributed system, access to some servers in the system might need to be controlled, operations such as printing might need to be billed to the correct user, and execution of some types of operations might need to be restricted to authorized users.

One of the earliest and most-widely used authentication services is called *Kerberos*. Kerberos allows users in an insecure network to authenticate their identity using a standardized protocol and a trusted key distribution centre, taking responsibility for the authentication of users and controlling their access to services within its *realm* (part of the system for which it has authentication responsibility). A Kerberos key distribution centre is logically divided into two parts, although in a small system both might reside on the same server machine:

authentication server which has a secret (master) key for secure communication with each user in its realm and which is responsible for checking user credentials to provide authentication, typically listening on port 88,

ticket granting server which the user consults once its identity has been authenticated to obtain authorization to use a service provided by a server in the realm, typically listening on port 749.

The protocol used by Kerberos follows an enhanced version of the steps from Section 1.6. The user obtains a *ticket granting ticket* (TGT) from the authentication server once its identity has been authenticated, which it can present to the ticket granting server to receive a *service granting ticket* (SGT) for a particular service in the realm, authorizing the user to access that service during a specified time period.

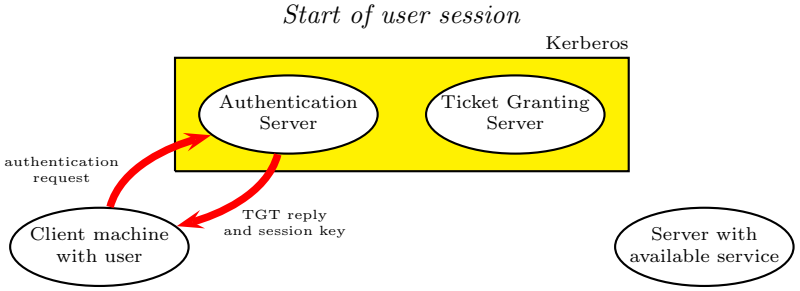
When a user logs on to the system via a client machine the client obtains the secret master key from the user for decrypting responses from the authentication server (such as by using password based encryption with a user-supplied password). The client sends a plaintext message to the authentication server requesting a ticket granting ticket to be produced for the user which vouches for the user's identity, including:

- options for requesting the TGT to have particular properties, such as being renewable or be postdated,
- an identifier for the user,
- the name of the realm,
- an identifier for the ticket granting server that will be used,
- a requested validity time period for the TGT,
- a nonce value.

The authentication server checks whether the user is known to it and if so responds to the client with a TGT suitably encrypted for the ticket granting server, and with a session key for the client communication with the ticket generating server during the lifetime of the TGT,

including:

- the name of the realm,
- an identifier for the user,
- the TGT which holds the session key, the name of the realm, an identifier for the user, the network address of the client, and validity period of the TGT, all encrypted by the authentication server for the ticket granting server using a secret key,
- the session key, the validity period of the TGT, the same nonce value, the name of the realm, and an identifier for the ticket granting server, all encrypted by the authentication server for the user using the master key.



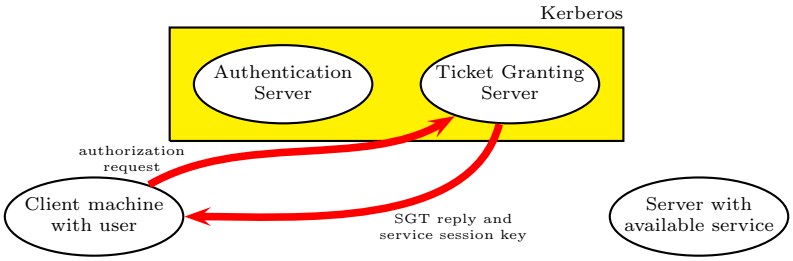
The ticket granting ticket can then be used by the client as proof of the user's authenticity during the remainder of the session or until the TGT expires. This avoids the need for the user to repeatedly enter its log on password during a single session, providing a feature known as *single sign-on*. Whenever the user tries to use a new service during that session the client presents the TGT to the ticket granting server, requesting a SGT to authorize its access to that service, including with the request:

- options for requesting the SGT to have particular properties,
- an identifier for the service,
- requested validity time period for the SGT,
- a nonce value,
- the TGT from the authentication server,
- an *authenticator* for the request, consisting of an identifier for the user, the name of the realm, and a timestamp, all encrypted by the client using the session key.

The ticket granting server decrypts the TGT to check its validity, determines the session key, and so can decrypt the authenticator to verify the authenticity of the sender. If the ticket granting server determines that the user should be authorized to use the service, then it responds to the client, including:

- the name of the realm,
- an identifier for the user,
- the SGT which holds a new session key for the client communication with the service, the name of the realm, an identifier for the user, the network address of the client, and validity period of the SGT, all encrypted by the ticket granting server for the service using a secret key,
- the session key for the client communication with the service, the validity period of the SGT, the same nonce value, the name of the realm, and an identifier for the service, all encrypted by the ticket granting server using its session key with the client.

First usage of a particular service during session



Once the client has received a service granting ticket for a particular service it can present it to the service with each request along with an authenticator for the request (which can optionally include an alternative session key chosen by the client, and a starting sequence number to detect replay attacks during the exchange). When the SGT expires, the client can request another SGT from the ticket granting server so long as it still has a valid TGT. This process is repeated for each new service the user requests.

The *Java Authentication and Authorization Service* (JAAS) is an API included with Java Standard Edition to support the authentication of users and their authorization to execute security-sensitive operations. It is a flexible framework found in the package `javax.security.auth` and subpackages of it, allowing a client to authenticate a user independently of the type of JAAS-compatible authentication service actually used in the system (such as Kerberos, a smart card system, or a biometric authentication system). A user or process that requires authentication is represented by the `Subject` class, and each identity that a `Subject` has, such as a user name within a realm or a unique identity number, is represented by the `Principal` interface (which is implemented by classes such as `KerberosPrincipal`). The class `SimplePrincipal` demonstrates a simple `Principal` that represents a user name given by a string.

Java Authentication and Authorization Service

```
/**
   A class that represents a simple JAAS Principal which provides a String name as an
       identity for a Subject
   @see SimpleLoginModule.java
*/
package simplejaasmodule;

import java.security.Principal;

public class SimplePrincipal implements Principal
{
   private String name;

   public SimplePrincipal(String name)
   { if (name == null)
         throw new IllegalArgumentException("No name for subject");
      this.name = name;
   }

   public boolean equals(Object another)
   { if (another==null || !(another instanceof SimplePrincipal))
         return false;
      else
      { SimplePrincipal principal = (SimplePrincipal)another;
         return name.equals(principal.getName());
      }
   }

   public String getName()
   { return name;
   }

   public int hashCode()
```

```
   { return name.hashCode();
   }

   public String toString()
   { return "SimplePrincipal with name " + name;
   }
}
```

A JAAS-compatible authentication service provides an implementation of the `LoginModule` interface, authenticating in a way suitable for that type of service. Authentication with a `LoginModule` is performed in two steps. Firstly, the `login` method is used to attempt authentication of a `Subject`, using a provided `CallbackHandler` to communicate with the user (such as to prompt for a user name and password and obtain the response). As the authentication process could require authentication by several successive `LoginModule`, any of which might fail to authenticate the user, a `LoginModule` has an `abort` method to abort the authentication (which gets called if authentication fails in another `LoginModule`), and a `commit` method to add a suitable `Principal` to the `Subject`. As an example, the class `SimpleLoginModule` performs authentication of a user via a `NameCallback` and a `PasswordCallback`.

JAAS Login

```
/**
   A class that represents a simple JAAS LoginModule which performs the authentication
       of a Subject via a specified CallbackHandler and assigns a SimplePrincipal to
       the Subject if the authentication is committed
   @author Andrew Ensor
*/
package simplejaasmodule;

import java.io.IOException;
import java.util.Arrays;
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class SimpleLoginModule implements LoginModule
{
   // subject being authenticated
   private Subject subject;
   // handles communication with user in application-specific way
   private CallbackHandler callbackHandler;
   //credential sharing across multiple LoginModule for single sign on
   private Map<String,?> sharedState;
   // configuration options for this LoginModule (allows debug=true)
   private Map<String,?> options;
   private boolean debugEnabled;
   private SimplePrincipal principal;
   private boolean committed = false;
   ...

   public void initialize(Subject subject,
      CallbackHandler callbackHandler, Map<String,?> sharedState,
      Map<String,?> options)
   { this.subject = subject;
      this.callbackHandler = callbackHandler;
      this.sharedState = sharedState;
      this.options = options;
```

```
    if (options.get("debug").toString().equalsIgnoreCase("true"))
       debugEnabled = true;
 }
 public boolean login() throws LoginException
 { principal = null;
    committed = false;
    if (callbackHandler == null)
       throw new LoginException("No callback handler provided");
    // prepare callbacks to pass to callbackHandler
    NameCallback nameCallback = new NameCallback("user name:");
    PasswordCallback passwordCallback
       = new PasswordCallback("password:", false);
    Callback[] callbacks = {nameCallback, passwordCallback};
    try
    { callbackHandler.handle(callbacks);
    }
    catch (IOException e)
    { throw new LoginException(e.toString());
    }
    catch (UnsupportedCallbackException e)
    { throw new LoginException(e.toString());
    }
    // extract the credentials obtained by callbackHandler
    String userName = nameCallback.getName();
    char[] tempPassword = passwordCallback.getPassword();
    if (tempPassword == null)
       tempPassword = new char[0];
    // copy the password to another array before it gets cleared
    char[] password = new char[tempPassword.length];
    System.arraycopy(tempPassword, 0, password, 0,
       tempPassword.length);
    passwordCallback.clearPassword();
    // output debugging information
    if (debugEnabled)
    { System.out.print("User entered:" + userName +
          " with password:");
       for (int i=0; i<password.length; i++)
          System.out.print(password[i]);
       System.out.println();
    }
    if (validate(userName, password))
    { principal = new SimplePrincipal(userName);
       return true;
    }
    else
    { // don't wait for garbage collection to clear the password
       Arrays.fill(password, ' ');
       return false;
    }
 }
 // helper method that tries to validate the user name and password
 // note in practice this would be checked in a secure database
 private boolean validate(String name, char[] password)
 { String validName = "Jack";
    char[] validPassword = {'c','h','a','n','g','e','i','t'};
    boolean validated = name.equals(validName) &&
       password.length==validPassword.length;
    if (validated)
    { for (int i=0; i<password.length; i++)
          if (password[i] != validPassword[i])
             validated = false;
    }
    // output debugging information
```

```
      if (debugEnabled)
         System.out.println("Authenticated user:" + validated);
      return validated;
   }

   public boolean abort() throws LoginException
   { if (principal == null)
         // own login failed so abort login is not performed
         return false;
      if (committed)
         // all LoginModule login succeeded but one's commit failed
         logout();
      else // another LoginModule login failed
         principal = null;
      return true; // abort was successful
   }

   public boolean commit() throws LoginException
   { if (principal == null)
         // own login failed so can not commit login
         return false;
      if (!subject.getPrincipals().contains(principal))
      { subject.getPrincipals().add(principal);
         if (debugEnabled)
            System.out.println("SimplePrincipal added to subject");
      }
      principal = null;
      committed = true;
      return true;
   }

   public boolean logout() throws LoginException
   { subject.getPrincipals().remove(principal);
      principal = null;
      committed = false;
      return true;
   }
}
```

It should be remembered however that a `LoginModule` would usually be implemented by a JAAS provider (such as a Kerberos provider) rather than by an application developer. As there might be several alternative authentication services available in a system, the `LoginModule` can all be specified in a configuration file, from which a client application can choose which configuration to use for authentication.

JAAS login configuration

```
/**
   JAASExample.config login configuration file with some
   configurations for a hypothetical distributed system
*/

SimpleLogin
{ simplejaasmodule.SimpleLoginModule required debug=true;
};

KerberosLogin
{ com.sun.security.auth.module.Krb5LoginModule required;
};

BiometricLogin
{ com.csp.jaasmodule.BioLoginModule sufficient;
```

```
   com.csp.jaasmodule.JavaCardLoginModule required matchOnCard="true";
};
```

The name of this configuration file can be specified either when the client application is executed:

```
java -Djava.security.auth.login.config==JAASExample.config...
```

or else in the Java security properties file `java.security` with the following lines:

```
login.configuration.provider=com.csp.JAASprovider
login.config.url.1=path/JAASExample.config
```

A client application can use JAAS to authenticate a user regardless of which authentication service is deployed in the system. First, a `CallbackHandler` is prepared to handle the actual communication with the user in a way suitable for the application (such as via `System.in` and `System.out` or instead via a GUI). Then the client creates a `LoginContext` which is used to log in and log out, specifying the name (given in the configuration file) of the login configuration to use, and a `CallbackHandler` to use during the authentication:

```
CallbackHandler callbackHandler = ...;
LoginContext lc
   = new LoginContext("SimpleLogin", callbackHandler);
lc.login();
Subject subject = lc.getSubject(); // authenticated subject
...
lc.logout();
```

The `LoginContext` takes care of locating the appropriate `LoginModule` specified in the configuration file and performs the necessary authentication(s).

Once a `Subject` has been obtained it can optionally be used to call a `PrivilegedAction`, which has security-sensitive operations (such as file write access) in a `run` method. In a security conscious system applications should not be allowed to run unrestricted (the default for a local Java application), but instead have a *fine-grained security* as specified by a *security policy* (typically determined by a system administrator). As described further in Section 5.2 a security policy file consists of a series of `grant` statements, each with optional signer field, a JAR file or URL codebase, and a principal, that together specify the code and/or user to which the permissions apply:

```
grant signedBy alias codeBase uri Principal PrinClass name
{  permission SecurityPermissionClass Action;
   ⋮
   permission SecurityPermissionClass Action;
};
```

When the static `Subject` method `doAsPrivileged` is called, the `run` method of the `PrivilegedAction` is called using whatever security policies are in place for the specified subject.

As an example, the class `JAASExampleClient` demonstrates a simple client application that uses the class `JAASExampleCallbackHandler` to handle communication with the user. The client application uses the `"SimpleLogin"` configuration to attempt authentication of a user (if a Kerberos authentication server were available then this configuration could be swapped to `"KerberosLogin"`). It then tries to use that user's security permission as specified in the security policy file `JAASExample.policy` to perform a security-sensitive operation, attempting to write a file to the local drive. Although `JAASExampleCallbackHandler` uses a `Scanner` to handle the input of the user name, note that it is more much careful when obtaining the user's password, avoiding the creation of any string that might linger on the client machine, and erasing all details of the password beside that held in a single `char[]` array.

JAAS USER AUTHENTICATION

```
/**
   A class that demonstrates how JAAS can be used to authenticate a user and authorize
       whether the user can execute a PrivilegedAction
```

```
   @author Andrew Ensor
*/
import java.io.FileWriter;
import java.io.IOException;
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

public class JAASExampleClient implements PrivilegedAction<Boolean>
{
   // PrivilegedAction run method that contains security-sensitive
   // operation and has any specified generic return type
   public Boolean run()
   { try
      { FileWriter fw = new FileWriter("localfile.txt");
         fw.write("JAAS says hello on local drive");
         fw.close();
         return true; // worked
      }
      catch (IOException e)
      { System.err.println("IO Exception while writing file: " + e);
         return false; // failed
      }
   }

   public static void main(String[] args)
   { PrivilegedAction<Boolean> privilegedAction
        = new JAASExampleClient();
      CallbackHandler callbackHandler=new JAASExampleCallbackHandler();
      try
      { System.out.println("Creating login context");
         LoginContext lc
            = new LoginContext("SimpleLogin", callbackHandler);
         System.out.println("Logging in");
         lc.login();
         Subject subject = lc.getSubject(); // authenticated subject
         System.out.println("Executing security-sensitive operation");
         // execute the privileged action as specified subject with
         // no protection domains taken from current thread's context
         boolean success = (Boolean)Subject.doAsPrivileged(subject,
            privilegedAction, null);
         System.out.println("Operation " +(success?"was":"not")
            + " successful");
         System.out.println("Logging out");
         lc.logout();
      }
      catch (LoginException e)
      { System.err.println("Login Exception: " + e);
      }
      catch (SecurityException e)
      { System.err.println("Security Exception: " + e);
      }
   }
}
```

JAAS Callback Handler

```
/**
   A class that represents a simple JAAS CallbackHandler that just uses System.in and
       System.out to communicate with the user. Note a more elaborate CallbackHandler
```

```
      might eg use a GUI
   @see JAASExampleClient.java
*/
import java.io.InputStream;
import java.io.IOException;
import java.io.PushbackInputStream;
import java.util.Arrays;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

public class JAASExampleCallbackHandler implements CallbackHandler
{
   private Scanner keyboardInput;

   public JAASExampleCallbackHandler()
   { keyboardInput = new Scanner(System.in);
   }

   public void handle(Callback[] callbacks)
      throws IOException, UnsupportedCallbackException
   { if (callbacks == null)
        return; // no callbacks to handle
      for (int i=0; i<callbacks.length; i++)
      { if (callbacks[i] instanceof TextOutputCallback)
        { TextOutputCallback callback
             = (TextOutputCallback)callbacks[i];
          switch (callback.getMessageType())
          { case TextOutputCallback.INFORMATION:
               System.out.println(callback.getMessage());
               break;
            case TextOutputCallback.ERROR:
               System.out.println("ERROR:"+callback.getMessage());
               break;
            case TextOutputCallback.WARNING:
               System.out.println("WARNING:"+callback.getMessage());
               break;
            default:
               throw new IOException("Unsupported message type: "
                  + callback.getMessageType());
          }
        }
        else if (callbacks[i] instanceof NameCallback)
        { NameCallback callback = (NameCallback)callbacks[i];
          // prompt user for name and get name using Scanner
          System.out.print(callback.getPrompt());
          callback.setName(keyboardInput.nextLine());
        }
        else if (callbacks[i] instanceof PasswordCallback)
        { PasswordCallback callback=(PasswordCallback)callbacks[i];
          // prompt user for password and get using util method
          System.out.print(callback.getPrompt());
          callback.setPassword(readPassword(System.in));
        }
        else
          throw new UnsupportedCallbackException(callbacks[i],
            "Unsupported callback");
      }
   }
```

```
  // utility method that obtains a password from an InputStream
  // without storing the password in any String
  private char[] readPassword(InputStream is) throws IOException
  { final int INITIAL_CAPACITY = 128;
    char[] buffer = new char[INITIAL_CAPACITY];
    int numChars = 0; // number of characters currently in buffer
    boolean done = false;
    while (!done)
    { int ch = is.read();
      switch (ch)
      { case -1:
        case '\n':
          done = true;
          break;
        case '\r':
          int nextChar = is.read();
          if ((nextChar != '\n') && (nextChar != -1))
          { if (!(is instanceof PushbackInputStream))
              is = new PushbackInputStream(is);
            ((PushbackInputStream)is).unread(nextChar);
          }
          else
            done = true;
          break;
        default:
          if (numChars >= buffer.length)
          { // buffer full so allocate a new larger buffer
            char[] newBuffer = new char[buffer.length*2];
            System.arraycopy(buffer, 0, newBuffer, 0, numChars);
            Arrays.fill(buffer, ' '); // clear old buffer
            buffer = newBuffer;
          }
          buffer[numChars++] = (char)ch;
          break;
      }
    }
    if (numChars == 0)
      return null;
    // copy used portion of buffer to new buffer that gets returned
    char[] returnBuffer = new char[numChars];
    System.arraycopy(buffer, 0, returnBuffer, 0, numChars);
    Arrays.fill(buffer, ' '); // clear buffer
    return returnBuffer;
  }
}
```

Exercise 4.1 (Fine-Grained Security Permissions) *Test the class* `JAASExampleClient` *with and without a restricted security policy specified at runtime, and arrange for several users to be granted different security permissions.*

4.2 Electronic Mail Security

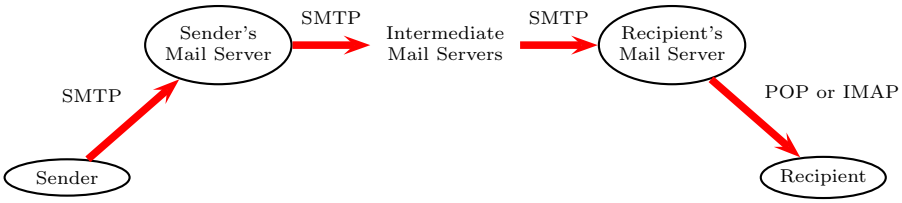
When a user sends an email from a email client program the message is first communicated via the *Simple Mail Transfer Protocol* (SMTP) to a mail server on the network (such as `mail.aut.ac.nz` on the AUT network or `smtp1.vodafone.net.nz` on the Vodafone network), which runs an SMTP service on port 25 and has responsibility for processing all incoming and outgoing email. The client starts the communication by identifying itself (`HELO`) and telling the server it has an email (`MAIL FROM`) to deliver to one or more recipients (`RCPT TO`), followed by the contents of the email (`DATA`). The contents of an email that adheres to the original *RFC 822* standard consists of a header (usually including fields such as `Date`, `From`, `To`, and `Subject`),

followed by a blank line and the text body of the message using 7-bit ASCII character encoding. It then indicates to the server that it has completed (a `.` followed by `QUIT`).

```
HELO clientURL
MAIL FROM: sender@aut.ac.nz
RCPT TO: recipient@aut.ac.nz
DATA
Date:  1/1/2007 12:30pm
From:  sender@whitehouse.com
To:  recipient@aut.ac.nz
Subject:  subject

Text Message
.
QUIT
```

The mail server responds to each directive with a numerical code and a message that indicates the success or failure of the requested operation. When a mail server receives an email message it checks whether it has an inbox for the message recipient, and if not it relays (forwards) the email to another mail server closer to the recipient's address.

When an email message arrives at the destination mail server it is stored in a file until the email client program of the recipient accesses it. The *Post Office Protocol* (POP) can be used to retrieve the mail for the recipient and remove it from the server. Instead, the *Internet Message Access Protocol* (IMAP) is a more advanced protocol that can be used. IMAP allows email messages to be stored on the mail server in folders rather than requiring them to be downloaded to a client. Thus IMAP makes the mailbox folders available across the network to the user but places a larger storage burden on the mail server.



Although email is one of the most heavily utilized network-based applications it is usually completely insecure, providing no confidentiality nor authentication of the user or message contents. By default email messages are passed unencrypted between servers, any of which could modify the message, replay an old message, or create a new message with a fabricated sender identity. Furthermore, each message includes the client URL and often information about the client email program, which raises privacy issues.

There are two standard alternative approaches for providing confidentiality and/or authentication of email, PGP and S/MIME. *Pretty Good Privacy* (PGP) relies on a *web of trust*, rather than a centralized CA, to decide the validity of a received public key, allowing the user to assign a level of trust to each public key it receives from other users. Since its introduction in 1991 PGP has been very popular amongst Internet users for providing secure personal email. It offers five services which get applied to a message in the following order:

authentication with a hash code created using SHA-1 which is then encrypted using either DSA or RSA with the sender's private key,

compression using ZIP compression to reduce the size of the transmitted message and eliminate repeated patterns in the plaintext,

confidentiality using either triple DES or CAST-128 (a 64-bit block cipher that uses a 128-bit key), with a (single-use) secret session key that has been generated specifically for encrypting that message,

transfer encoding using Base 64 encoding so that the compressed and encrypted message can be transmitted as text content via SMTP,

segmentation of long messages into several smaller segments which are mailed separately and

reassembled at the receiving end.

The content of a PGP message consists of three parts, a session key component, a signature component, and the message itself. Both the signature component and the message component are first compressed and then encrypted using the secret session key, and then the entire message content (including the session key component) gets encoded using Base 64 encoding. In case the sender or recipient might have several public/private key pairs in use the message includes a key identifer for the public key of each to help identify which key has been used for encryption (simply providing the least significant 64 bits of the public key).

Session key	Least significant 64 bits of recipient's public key Session key encrypted using recipient's public key
Signature	Timestamp Least significant 64 bits of sender's public key First 16 bits of hash code Hash code encrypted using sender's private key
Message	Filename Timestamp Message Data

The *Secure/Multipurpose Internet Mail Extension* (S/MIME) provides digital signatures and encryption for the standard MIME mail format, which itself extends the message format of RFC 822 to allow non-ASCII character sets, non-text attachments, and multi-part message bodies. The following MIME types can be specified in the `Content-type` header field of a MIME message:

application such as `application/java-archive` for JAR attachments and `application/msword` for MS Word documents,

audio such as `audio/x-midi` and `audio/x-wav` for audio data,

image such as `image/jpeg` and `image/png` for image data,

message such as `message/partial` for fragmentation of large messages and `message/external-body` for references to external content,

multi-part such as `multipart/mixed` for several independent message parts and `multipart/alternative` for several alternative forms of the same information, where the parts are separated by specified boundary strings,

text such as `text/plain` for plain unformatted ASCII text and `text/html` for HTML content,

video such as `video/mpeg` and `video/quicktime` for video data.

MIME also allows the transfer encoding scheme, such as `7bit` (for no encoding) or `base64`, to be specified in the `Content-Transfer-Encoding` header field. The S/MIME content type `application/pkcs7-mime` is used to specify a digital signature and/or encryption of a MIME message or part of it. Signatures are created using SHA-1 (or the less secure MD5) to generate a hash code that gets encrypted using the sender's private key and then Base 64 encoded. As in PGP, encryption is performed using triple DES (or possibly AES) with a single-use secret session key that has been generated specifically for encrypting that message, and the resulting ciphertext is referred to as *enveloped data*.

The *JavaMail* API is included as part of Java Enterprise Edition, but is also available as an optional API for Java Standard Edition. Although TCP can be used directly to communicate with a mail server for sending email via SMTP, and retrieving email via either POP or IMAP, JavaMail can be used to facilitate the process, hiding the details of each protocol. The JavaMail API requires the *JavaBeans Activation Framework* (JAF) API, which supports handling arbitrary blocks of data and determining their correct MIME types. Surprisingly, version 1.5 of JavaMail does not yet support PGP nor S/MIME (it is awaiting finalization of the S/MIME IETF specification), so encryption and authentication must either be added to messages or else a third-party extension to JavaMail used. The `Session` class in the `javax.mail` package defines a session with a mail server. Its static `getDefaultInstance` method is passed the mail server URL in a `Properties` object and an optional `Authenticator` which prompts for the user name and password as required (and which can be designed to have single sign-on capability),

returning a default (shared) mail session:

```
Properties properties = System.getProperties();
properties.put("mail.smtp.host", HOST); // or pop3 or imap
Authenticator authenticator = ...; // or null if not required
Session session = Session.getDefaultInstance(properties,
   authenticator);
```

To send a message a `MimeMessage` can be created from the session and the `Transport` method `send` used to send the message to the mail server via SMTP:

```
MimeMessage message = new MimeMessage(session);
message.setFrom(...);
message.addRecipient(Message.RecipientType.TO, ...);
... // set content of message
Transport.send(message);
```

To retrieve mail from a mail server a `Store` is used via either POP or IMAP:

```
Store store = session.getStore("pop3");
store.connect();
```

which provides access to the mailbox folders on the mail server. The example classes `EMailSender` and `EMailReceiver` demonstrate how JavaMail can be used to send a multipart message and retrieve the messages in the `INBOX` folder.

JavaMail API

```
/**
   A class that demonstrates the optional JavaMail API for sending email messages with
       multiple MIME parts in the body of the message. Note this class needs the
       optional JavaMail to compile, place the JAR file mail.jar in classpath
   @author Andrew Ensor
*/
import java.util.Properties;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import javax.mail.BodyPart;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMultipart;
import javax.mail.internet.MimeMessage;

public class EMailSender
{
   private static final String HOST = "smtp.aut.ac.nz";

   public static void main(String[] args)
   { Properties properties = System.getProperties();
      properties.put("mail.smtp.host", HOST);
      // obtain the sender and recipient email addresses
      Scanner keyboardInput = new Scanner(System.in);
      System.out.print("Please enter sender address:");
      String from = keyboardInput.nextLine();
      System.out.print("Please enter recipient address:");
      String to = keyboardInput.nextLine();
      System.out.print("Please enter file to attach:");
```

```
      String filename = keyboardInput.nextLine();
      // obtain shared default mail session
      Session session = Session.getDefaultInstance(properties, null);
      // create a message
      MimeMessage message = new MimeMessage(session);
      try
      { message.setFrom(new InternetAddress(from));
         message.addRecipient(Message.RecipientType.TO,
            new InternetAddress(to));
      }
      catch (AddressException e)
      { System.err.println
            ("Address exception while preparing addresses: " + e);
      }
      catch (MessagingException e)
      { System.err.println
            ("Messaging exception while preparing addresses: " + e);
      }
      try
      { message.setSubject("Greeting");
         // prepare a MIME part that will default to text/plain
         BodyPart textPart = new MimeBodyPart();
         textPart.setText("Hello EMail World");
         // prepare a MIME part whose content type determined by JAF
         BodyPart filePart = new MimeBodyPart();
         DataSource fileSource = new FileDataSource(filename);
         filePart.setDataHandler(new DataHandler(fileSource));
         filePart.setFileName(filename);
         // prepare a multipart message using the two body parts
         Multipart multipart = new MimeMultipart();
         multipart.addBodyPart(textPart);
         multipart.addBodyPart(filePart);
         message.setContent(multipart);
         // send the message
         Transport.send(message);
         System.out.println("Message sent");
      }
      catch (MessagingException e)
      { System.err.println("Messaging exception: " + e);
      }
   }
}
```

JavaMail API

```
/**
   A class that demonstrates the optional JavaMail API for receiving email messages
       from a POP or IMAP mail server. Note this class needs the optional JavaMail to
       compile, place the
   JAR file mail.jar in classpath
   @author Andrew Ensor
*/
import java.util.Properties;
import javax.mail.Authenticator;
import javax.mail.Folder;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.NoSuchProviderException;
import javax.mail.Session;
import javax.mail.Store;

public class EMailReceiver
```

```
{
   private static final String HOST = "pop.aut.ac.nz"; // AUT student

   public static void main(String[] args)
   { Properties properties = System.getProperties();
      properties.put("mail.pop3.host", HOST);
      // obtain shared default mail session
      Authenticator authenticator = new SimpleAuthenticator();
      Session session = Session.getDefaultInstance(properties,
         authenticator);
      // connect to mailbox store
      Store store = null;
      try
      { store = session.getStore("pop3"); // POP server
      }
      catch (NoSuchProviderException e)
      { System.err.println("Mail provider not available: " + e);
      }
      Folder folder = null;
      try
      { store.connect(); // use authenticator for user log in
         System.out.println("Connected to mail inbox");
         // obtain messages in specified folder
         folder = store.getFolder("INBOX");
         folder.open(Folder.READ_ONLY);
         Message[] messages = folder.getMessages();
         System.out.println("Messages currently in inbox:");
         for (int i=0; i<messages.length; i++)
         { Message message = messages[i];
            System.out.println("Message from " + message.getFrom()[0]
               + " with subject " + message.getSubject());
         }
      }
      catch (MessagingException e)
      { System.err.println("Messaging exception: " + e);
      }
      finally
      { try
         { if (folder != null)
               folder.close(false); // don't expunge deleted folders
            if (store != null)
               store.close();
         }
         catch (MessagingException e)
         {} // ignore
      }
   }
}
```

EMAIL AUTHENTICATOR

```
/**
   A class that provides a simple Authenticator for EMailReceiver. Note that this
       class does not obtain and store the password in a secure manner
   @see EMailReceiver.java
*/
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import javax.mail.Authenticator;
import javax.mail.PasswordAuthentication;

public class SimpleAuthenticator extends Authenticator
{
   private Scanner keyboardInput;
```

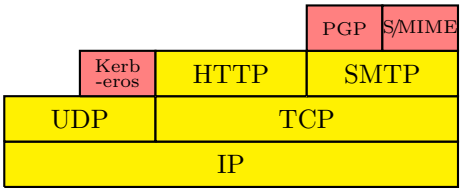
```
  public SimpleAuthenticator()
  { keyboardInput = new Scanner(System.in);
  }

  public PasswordAuthentication getPasswordAuthentication()
  { // prompt user for name and get name using Scanner
     System.out.print("Please enter user name:");
     String username = keyboardInput.nextLine();
     // prompt user for password and get using Scanner
     System.out.print("Please enter password:");
     String password = keyboardInput.nextLine(); // not secure here
     return new PasswordAuthentication(username, password);
  }
}
```

Exercise 4.2 (Adding a Message Signature) *Modify* `EMailSender` *so that it adds a digital signature as a MIME part to an email message.*

4.3 Web Security

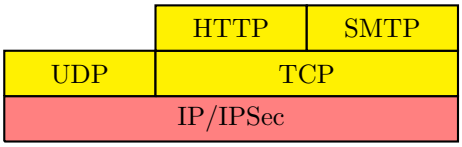
Using an authentication service such as Kerberos, or secure email via PGP or S/MIME, presumes that the communication in a system is insecure, and so security is provided at the application level. A complementary approach to ensuring security in a system is to secure the network communication channels, either at the network protocol or at the transport protocol level. By implementing security at the IP level, all network communication is made secure transparently to the applications in the system.

Application Level Security



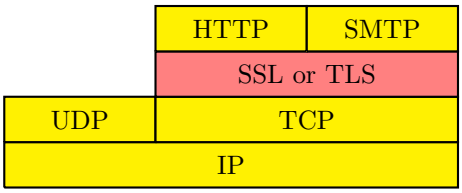
IP security (IPSec) provides authentication, confidentiality, and key management capabilities for IP packets (optional for IPv4 but mandatory for the newer IPv6). It is implemented as extension headers that follow the main IP header in each packet. Encryption is provided by a cipher such as triple DES or Blowfish, and authentication by HMAC with either MD5 or SHA-1 (but producing a 96 bit MAC by default). Secret keys are distributed using a version of Diffie-Hellman key exchange known as *Oakley key exchange*, which provides added security by using cookies to counter clogging attacks, nonces for replay attacks, and authentication for man-in-the-middle attacks.

Network Level Security



The *Secure Sockets Layer* (SSL) and its Internet standard version *Transport Layer Security* (TLS) provide confidentiality and authentication for applications that use TCP communication. It establishes an *SSL session* between a client and a server using a set of security parameters, which are shared for several secure *SSL connections* between the client and server. The parameters specify a session identifier, the algorithms used for compression and encryption, and hash functions used for MAC calculations. They also include a secure 384-bit *master secret* shared by both the client and server. This value is calculated from a 384-bit *pre-master-secret* (either by the client and sent to the server encrypted using the RSA public key of the server, or else via Diffie-Hellman key exchange) using a combination of MD5 and SHA-1, and is used to determine any secret keys and initialization vectors required during the session.

Transport Level Security



Both SSL and TLS use X.509 certificates to provide authentication. In order for the server to authenticate itself it must deliver a valid certificate to the client, for which the client can establish a certificate chain to a certificate that it already trusts, and then the server must prove that it is indeed the subject of the certificate by successfully decrypting something (such as the pre-master-secret), which can only be done by the holder of the private key. Optionally, the client too can be made to authenticate itself to the server using its certificate. The authentication steps are automated by the protocol so long as the server and client have appropriate certificates.

SSL and TLS process data before they get sent via TCP by breaking the data into fragments of up to 16384 bytes and applying the following steps to each fragment:

- firstly the fragment can optionally be compressed,
- a 128-bit or 160-bit MAC is calculated for the compressed data (TLS uses HMAC with either MD5 or SHA-1, whereas SSL uses an older variant of HMAC), and is appended to the compressed fragment,
- the entire fragment (including the MAC) is then encrypted using a block cipher such as AES, DES, or triple DES (padding the fragment so that the final padding byte gives the number of bytes of padding added), or else using the RC4 stream cipher (with no padding required),
- a five byte SSL header is prepended to the fragment to indicate the type of SSL fragment, the major protocol version (currently `3` for both SSL and TLS), the minor protocol version (`0` for SSL and currently `1` for TLS), and two bytes that give the length of the fragment.

The resulting fragments are then communicated via TCP. There are four types of SSL fragment that can be communicated:

application data where the data in the fragment has originated from an application that is using an application level protocol such as HTTP or SMTP for communication,

change cipher where the data is a single fixed byte `01`, indicating that the cipher suite for the connection is to be updated with the next pending values,

alert where the data consists of two alert bytes, the first byte a `01` (a warning) or `02` (a fatal error which results in the connection being terminated), and the second byte a code to indicate the specific alert (such as a warning alert that a certificate was not provided during authentication, or a fatal alert about an incorrect MAC),

handshake where the client and server negotiate details for establishing the connection, consisting of a byte to indicate one of 10 possible types of message, three bytes to indicate the length of parameters, followed by the parameters (such as an encrypted key or an X.509 certificate).

The handshaking is used to have the server authenticate itself to the client (and possibly also the client to the server), create the master secret, and establish which encryption, compression, and MAC algorithms will be used during communication. Handshaking proceeds in four phases:

Establish security capabilities The client initiates the handshake by sending a `client_hello` message requesting a protocol version, preferred security algorithms, and a nonce value, which the server responds to with a `server_hello` message specifying its choice of algorithms, and its own nonce value.

Server authentication and key exchange The server then sends a `certificate` message holding a certificate for the server, then (if required by the encryption algorithm) a `server_key_exchange` containing server key parameters and a signature using the nonce values, an optional `certificate_request` message requesting a certificate from the client, followed by a `server_done` message.

Client authentication and key exchange The client continues by sending a `certificate` message if requested, then a compulsory `client_key_exchange` containing client key parameters and a signature, optionally followed by a `certificate_verify` message that uses the client's private key to encrypt a hash code of the handshake message (to prove that the client does possess the private key corresponding to the public key provided in the certificate).

Finish handshake Finally the client and server notify each other to start using the cipher. The client sends a `change_cipher_spec` message and a `finished` message, and the server responds with a `change_cipher_spec` message and its own `finished` message.

The *Java Secure Socket Extension* (JSSE) is an API included with Java Standard Edition that provides an implementation of SSL/TLS. It includes functionality for data encryption, server authentication, message integrity, and optional client authentication, hiding the protocol details for cipher suite negotiation and the initial handshaking. A keystore that holds trusted certificates used to verify received certificates during handshaking is called a *truststore*. The keystore and/or truststore to be used by JSSE can either be specified as command line arguments when the client or server is run:

```
java -Djavax.net.ssl.trustStore=mytruststore.jks
   -Djavax.net.ssl.trustStorePassword=mytrustpw
   -Djavax.net.ssl.keyStore=mykeystore.jks
   -Djavax.net.ssl.keyStorePassword=mystorepw WhateverClass
```

or else specified via the `System` method `setProperty`:

```
System.setProperty("javax.net.ssl.trustStore", "...");
System.setProperty("javax.net.ssl.trustStorePassword","...");
System.setProperty("javax.net.ssl.keyStore", "...");
System.setProperty("javax.net.ssl.keyStorePassword", "...");
```

Once JSSE is provided with a truststore an SSL/TLS client can be implemented similarly to an (insecure) TCP client by creating an `SSLSocket` via an `SSLSocketFactory`:

```
SSLSocketFactory sslFactory
   = (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket socket
   = (SSLSocket)sslFactory.createSocket(HOST_NAME,HOST_PORT);
socket.startHandshake(); //optional,happens anyway when flush
...
socket.close();
```

Also, once JSSE is provided with a keystore an SSL/TLS server can use an `SSLServerSocketFactory` to create an `SSLServerSocket`:

```
SSLServerSocketFactory sslServerFactory
   = (SSLServerSocketFactory)
   SSLServerSocketFactory.getDefault();
SSLServerSocket serverSocket = (SSLServerSocket)
   sslServerFactory.createServerSocket(PORT);
SSLSocket socket = (SSLSocket)serverSocket.accept();
socket.setWantClientAuth(true); // either none, want, or need
...
socket.close();
serverSocket.close();
```

Optionally, the cipher algorithms negotiated during the handshaking can be restricted by specifying the allowable algorithms to either the `SSLServerSocket` or `SSLSocket`:

```
String[] enabledSuites = {"TLS_RSA_WITH_AES_128_CBC_SHA"};
socket.setEnabledCipherSuites(enabledSuites);
```

As an example, the classes `SSLEchoClient` and `SSLEchoServer` demonstrate how SSL/TLS can be used to ensure confidentiality and authentication of communication between a TCP client and server. The server uses a keystore called `jssekeystore.jks` to select a public/private key pair and send a certificate for the public key. The client uses a truststore called `jssetruststore.jks` to check the received certificate against its own trusted certificates. The following three command line statements can be used to create the keystore and generate an RSA key pair, export a corresponding self-signed certificate, and import the certificate to a new truststore (so that it is immediately verified without actually building a certificate chain to a CA):

```
keytool -genkey -alias ssltest -keyalg RSA
```

```
   -keystore jssekeystore.jks
keytool -export -alias ssltest -file ssltest.cer
   -keystore jssekeystore.jks
keytool -import -alias ssltest -file ssltest.cer
   -keystore jssetruststore.jks
```

The communication between the client and server during handshaking can be displayed by enabling the *dynamic debug tracing* support, running the server via the command line statement:

```
java -Djavax.net.debug=SSL,handshake,data,trustmanager
   SSLEchoServer
```

As JSSE uses the cryptographic capabilities of the JCA, it can be configured to use a cryptographic token such as a smart card or a hardware cryptographic accelerator via PKCS#11. To use a smart card as a keystore or truststore the `javax.net.ssl.keyStoreType` or `javax.net.ssl.trustStoreType` system property should be set to `"pkcs11"`, and the `javax.net.ssl.keyStore` or `javax.net.ssl.trustStore` system property to `"NONE"`. Also, JSSE has support for Kerberos cipher suites provided that both the SSL/TLS client and server have accounts with the Kerberos authentication server, in which case JAAS and a Kerberos `LoginModule` can be used to obtain the necessary Kerberos credentials.

The `java.net` class `URL` supports *HTTP over SSL/TLS* (HTTPS), for using HTTP communication over SSL/TLS for secure communication with web servers (using default port 443 instead of the HTTP default port 80):

```
URL pageURL = new URL("https:...");
HttpsURLConnection conn
= (HttpsURLConnection)pageURL.openConnection();
InputStream is = pageURL.openStream();
```

The class `HTTPSExample` is a simple example of establishing an HTTPS connection. Note that besides setting system properties for the truststore it must also set system proxy properties if it accesses a web server from behind a firewall (discussed in Section 5.3).

Exercise 4.3 (HTTPS With Generic Connection Framework) *Use the Java Micro Edition class* `Connector` *to open an* `HttpsConnection` *to a web server and obtain details of the server certificate.*

SSLEchoServer

```
/**
   A class that represents a client that repeatedly sends any keyboard input securely
       to an SSLEchoServer until the user enters DONE
   @see SSLEchoServer.java
*/
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

public class SSLEchoClient
{
   public static final String HOST_NAME = "localhost";
   public static final int HOST_PORT = 8890; // host port number
   public static final String DONE = "done"; // terminates echo
   ...

   public void startClient()
   { Scanner keyboardInput = new Scanner(System.in);
      System.setProperty("javax.net.ssl.trustStore",
        "jssetruststore.jks");
```

```
      System.setProperty("javax.net.ssl.trustStorePassword",
         "changeit");
      SSLSocketFactory sslFactory
         = (SSLSocketFactory)SSLSocketFactory.getDefault();
      SSLSocket socket = null;
      try
      { socket
           = (SSLSocket)sslFactory.createSocket(HOST_NAME,HOST_PORT);
         System.out.println("Starting handshake");
         socket.startHandshake(); //optional,happens anyway when flush
      }
      catch (IOException e)
      { System.err.println("Client could not make connection: " + e);
         System.exit(-1);
      }
      PrintWriter pw = null; // output stream to server
      BufferedReader br = null; // input stream from server
      ...
   }

   public static void main(String[] args)
   { SSLEchoClient client = new SSLEchoClient();
      client.startClient();
   }
}
```

SSL/TLS TCP MESSAGE ECHO

```
/**
   A class that represents a server that continually echos any SSL/TLS TCP message
       securely back to the client until it receives DONE
   @author Andrew Ensor
*/
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.net.SocketTimeoutException;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;

public class SSLEchoServer
{
   private boolean stopRequested;
   public static final int PORT = 8890; // some unused port number
   public static final String DONE = "done"; // terminates echo

   public SSLEchoServer()
   { stopRequested = false;
   }

   // start the server if not already started and repeatedly listen
   // for client connections until stop requested
   public void startServer()
   { stopRequested = false;
      System.setProperty("javax.net.ssl.keyStore", "jssekeystore.jks");
      System.setProperty("javax.net.ssl.keyStorePassword", "changeit");
      SSLServerSocketFactory sslServerFactory
         =(SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
```

```
      SSLServerSocket serverSocket = null;
      try
      { serverSocket = (SSLServerSocket)
            sslServerFactory.createServerSocket(PORT);
         serverSocket.setSoTimeout(2000); // timeout for accept
         System.out.println("Server started at "
            + InetAddress.getLocalHost() + " on port " + PORT);
      }
      catch (IOException e)
      { System.err.println("Server can't listen on port: " + e);
         System.exit(-1);
      }
      while (!stopRequested)
      { // block until the next client requests a connection
         // or else the server socket timeout is reached
         try
         { SSLSocket socket = (SSLSocket)serverSocket.accept();
            System.out.println("Secure connection made with "
               + socket.getInetAddress());
            // start an echo with this connection
            EchoConnection echo = new EchoConnection(socket);
            Thread thread = new Thread(echo);
            thread.start();
         }
         catch (SocketTimeoutException e)
         { // ignore and try again
         }
         catch (IOException e)
         { System.err.println("Can't accept client connection: " + e);
            stopRequested = true;
         }
      }
      try
      { serverSocket.close();
      }
      catch (IOException e)
      { // ignore
      }
      System.out.println("Server finishing");
   }

   // stops server AFTER the next client connection has been made
   // or timeout is reached
   public void requestStop()
   { stopRequested = true;
   }

   // driver main method to test the class
   // note that it doesn't call requestStop so main does not exit
   public static void main(String[] args)
   { SSLEchoServer server = new SSLEchoServer();
      server.startServer();
   }

   // inner class that represents a single echo connection
   private class EchoConnection implements Runnable
   {
      ...
   }
}
```

HTTPS CONNECTION

```
/**
   A class that demonstrates how to open an HTTPS connection to a web server and
       obtain the response
   @author Andrew Ensor
*/
...
public class HTTPSExample
{
   public static void main(String[] args)
   { // use the cacerts keystore included with Java Runtime
      Scanner keyboardInput = new Scanner(System.in);
      System.setProperty("javax.net.ssl.trustStore",
         "c:/j2sdk/jre/lib/security/cacerts");
      System.setProperty("javax.net.ssl.trustStorePassword",
         "changeit");
      // set the proxy host and port number for AUT's firewall
      // note this is not needed if no firewall present
      System.setProperty("https.proxySet","true");//true if using proxy
      System.setProperty("https.proxyHost", "cache.aut.ac.nz"); // AUT
      System.setProperty("https.proxyPort", "3128"); // AUT specific
      // obtain the URL for https
      String urlName = null;
      ...
      try
      { URL pageURL = new URL(urlName);
         // set connection and read timeouts for the connection
         HttpsURLConnection conn
            = (HttpsURLConnection)pageURL.openConnection();
         conn.setConnectTimeout(5000); // 5000ms
         conn.setReadTimeout(5000); // 5000ms
         // create an input stream and read all the lines
         InputStream is = pageURL.openStream();
         BufferedReader br = new BufferedReader
            (new InputStreamReader(is));
         String line = br.readLine();
         while (line != null)
         { System.out.println("Received: " + line);
            line = br.readLine();
         }
         br.close();
      }
      catch (MalformedURLException e) // not valid protocol
      { System.err.println("Invalid protocol or URL: " + e);
      }
      catch (IOException e)
      { System.err.println("IO Exception: " + e);
      }
   }
}
```

Chapter 5

System Security

5.1 Intruders and Malicious Software

An *intruder* is an individual who gains unauthorized access to a computer system, either by exploiting the account of a legitimate user (a *masquerader*), being a legitimate user who accesses the system in a way that is unauthorized (a *misfeasor*), or else by obtaining some type of administration access to the system which can be used to evade detection (a *clandestine user*).

Some systems might include a *back door* which provides a (non-obvious) way to bypass the normal authentication requirements of the system. Back doors are sometimes included by a programmer during development and testing of a system, either to speed up testing or else to provide access in case there might be a problem with the authentication process. In other systems, back doors might occur due to subtle security flaws resulting from a developer's failure to grasp complex security issues. For instance, an intruder might gain access to security policy settings or be able to forge a authentication ticket. Back doors can also be set up by malicious software such as viruses or worms when the access controls for a system have been compromised. Instead, a malicious compiler could insert a back door into an application, which would be difficult to detect as no evidence would appear in any source code of the application.

If a back door to a system cannot be found then the intruder must somehow gain access via the authentication procedures of the system. Typically this is achieved by the intruder obtaining the password of a legitimate user. Attempts to guess a password during user authentication can be easily detected by a system, which can foil a brute force attempt by insisting that each authentication attempt take a certain time period, and/or disconnecting or temporarily invalidating a user account after several failed attempts. Although a system may try to prevent access to its files that store user passwords, this cannot be relied on as a security lapse might make such a file available within the system, an administrator with access to all files might use the same password on another system which has already been compromised, or a misfeasor might be able to impersonate software which is granted read access to a password file for the purpose of authentication. To prevent actual user passwords from becoming available to a potential intruder, they are never stored as plaintext in a file. Instead, either only their hash codes or some encrypted form of them (such as by password based encryption) is ever stored:

Host	*User*	*Password*
156.62.%	student	`6E41903A2A2A67A4`
156.62.%	admin	`175EA45B7CDF2B4A`

The security of the passwords in a password file can be improved by assigning a random salt to each user, making the salts available to authentication software, and using them in the calculation of the encrypted passwords. Thus two users which happen to have the same password in a system would not have the same encrypted form in the file, and if the password file were obtained by an intruder any attempt to crack a password would have to focus on one user at a time.

One significant problem with password-based authentication is that many users choose pass-

words that can be easily cracked by guessing, often choosing a password based on their own or someone else's name, relevant personal information such as an address, dictionary words, or simple variations of these using a mixture of upper and lower case. If a copy of the password file for a system could be obtained then millions of potential passwords could be checked by the intruder off-line. Studies have shown that up to 25% of user-chosen passwords can be cracked using password lists commonly available to potential intruders.

Password cracking can be countered by having a system insist on *password selection rules*, where a password selected by a user is scrutinized for its suitability before it gets accepted. Some common rules are to insist that every password be at least eight characters long (to counter a brute-force attack), that it not appear in a common password list nor in a dictionary, include a mix of upper and lower case and include at least one non-letter character. To avoid storing an enormous list of non-allowable passwords a *Bloom filter* could be used, where several (short) hash functions are used to calculate various hash codes for a password. Each hash function has an associated hash table with a bit allocated for each of the possible hash codes it can produce. A bit is set in each table corresponding to the hash code for each non-allowable password. When a submitted password is scrutinized its hash codes are calculated, and if the corresponding bit in every one of the tables is set then the password gets rejected.

The introduction of smart cards and/or biometric identification has greatly improved the security of password-based authentication, reducing many of the security risks associated with passwords and password files. For instance a smart card can render itself invalid if incorrect pin are repeatedly provided, it can ensure all communication with a host is confidential and authenticated, and biometric information such as fingerprint or facial recognition can be very difficult for an intruder to forge.

Although intrusion prevention is very important, a system should also have measures in place for detecting intruders once they have successfully entered, and take action to minimize the effects of the intrusion. One technique for intrusion detection is to store some information about the typical behaviour of each user, such as what applications and files are normally used by that user, and when that user typically logs in and out of the system. Then during each session the behaviour of a user is compared with their past behaviour and anything unusual (such as after hours access) might indicate that an intruder is masquerading as that user. Another technique is to enforce rules on proper user behaviour where multiple unusual actions might indicate an intruder, such as the copying of system files, attempts to access the password file, or writing to another user's files. Both these techniques rely on the system maintaining a log of significant events, such as authentication and authorization requests, unusual network communication, and access to/from outside the network. However, intruders that gain administration access rights might be able to modify the log files and hide their own actions. A security-conscious system might include *honeypots*, which are machines or even entire fabricated subsystems designed to appear attractive to intruders but which would be unknown to typical users, luring intruders away from actual critical parts of the system and noting their behaviour while countermeasures are taken against them.

Unfortunately, probabilistic measures for determining intruders suffer from *base-rate fallacy*, where the number of false alarms can be unsatisfactorarily high. Bayes' Theorem is a result from probability theory for mutually exclusive events $E_1, E_2, \ldots, E_n$, one of which must occur (so the sum of their probabilities is 1). For an arbitrary event A it states that:

$$\text{Prob}(E_i \text{ given } A) = \frac{\text{Prob}(A \text{ given } E_i) \cdot \text{Prob}(E_i)}{\sum_{j=1}^{n} \text{Prob}(A \text{ given } E_j) \cdot \text{Prob}(E_j)}.$$

To illustrate base-rate fallacy suppose a test is devised for detecting intruders, where an actual intruder is correctly detected by the test 95% of the time, so Prob(detected given intruder) = 0.95, and a legitimate user is also correctly detected 95% of the time. Suppose the proportion of intruders compared to legitimate users is low, such as only 2% of users are actually intruders. Bayes' Theorem can be used to give the probability that a user which has been detected as being an intruder is actually legitimate:

$$\begin{aligned}
&\text{Prob(legitimate given detected)} \\
&= \frac{\text{Prob(detected given legitimate)} \cdot \text{Prob(legitimate)}}{\text{Prob(detected given legit)} \cdot \text{Prob(legit)} + \text{Prob(detected given intruder)} \cdot \text{Prob(intruder)}} \\
&= \frac{0.05 \times 0.98}{0.05 \times 0.98 + 0.95 \times 0.02} \\
&= 0.7205,
\end{aligned}$$

so about 72% of the detected intruders are actually legitimate users.

Rather than intruders themselves trying to gain access into a system they might instead try to introduce malicious software. Some categories of malicious software are:

Logic bomb which contains code set to activate when certain conditions are met, such as at a particular time or when a certain application is utilized.

Trojan horse which is software that appears useful to a user but which contains some hidden code that performs an unexpected function.

Key logger which captures and stores keystrokes that are made on a machine, enabling an attacker to catch the password and confidential details typed by a user.

Zombie which is software that secretly takes over a machine on a network, and which can be used by an attacker to launch attacks on other machines. Zombies are used by attackers to make it more difficult to trace their actual location and to have multiple machines at their disposal for attacking a target. To find a new zombie an attacker might scan a network for vulnerable machines or else use a known hit-list of poorly protected sites.

Virus which is software that embeds itself in an executable program so that whenever the program is executed by a user the virus code is also performed. A virus might be introduced into a system and mistakenly executed by an application that automatically executes macro commands attached to an e-mail or spreadsheet. It propagates itself by locating some accessible executable programs on the user's account and tries to rewrite them by adding its own commands to the start of the executable file, thus infecting further files and possibly lodging itself inside operating system software.

Worm which can replicate itself automatically (without requiring any action to be performed by a user) and send copies of itself over a network. A worm actively seeks out more machines to infect (for example, by using a user's e-mail contact list or by randomly generating IP addresses), and might replicate itself using e-mail or via some remote execution or remote login capability of the system. It might disguise itself as a system process running in the background, actively try to crack passwords, communicate with the command interpreter of the operating system, download further versions of itself, or exploit flaws in system software (for instance, several worms have very successfully exploited security flaws in Microsoft Internet Information Server software). Worms are often used to install back doors on machines, so that they can be used as zombies for sending spam or to exploit a possible buffer overflow vulnerability of the operating system.

In a *distributed denial of service* (DDOS) attack many zombies are used to swamp a target machine with network communication in order to disrupt the functionality of the target and its availability to legitimate clients. The communication might be TCP/IP synchronization/initialization packets with false IP return address, or instead ECHO packets sent to innocent *reflectors* giving the target machine as the return address, so each reflector then sends an ECHO response to the target. To counter such attacks a potential target needs to have backup resources available in the event of an attack, be able to quickly identify a potential DDOS attack, and filter received communication to minimize its effects.

Early antivirus software tried to detect malicious software by scanning for occurrences of known virus code, but virus and worms were quickly developed by attackers that modified themselves to evade detection, such as by randomly changing some code or else encrypting parts of themselves using a randomly selected key (thus making each virus unique and potentially mutating with each execution). Some viruses compressed the original executable file before they embedded themselves, decompressing the file again when executed, thus making it impossible to tell from file sizes whether or not a file was infected. Hence antivirus software was developed which could scan executable files and inspect their code, such as checking whether they started by

performing some decryption or decompression, and if so find the key embedded within the malicious software and expose the virus code. Another approach was to calculate a hash code for each executable file and securely store it, or use a digital signature, so that modified files could be identified. Modern antivirus software can track the actions being performed by an executable program as it runs to catch suspicious activity (called a *activity trap*). Instead, an executable might be initially run using a software CPU emulator when started to try catching any virus before it gets a chance to cause damage to the system. The longer the executable is left in the emulator the greater the chance that virus activity gets detected, but the slower the execution of valid code. Such features are sometimes incorporated into distributed system software, with antivirus software running in the background on each machine to detect suspicious activities and unexpected background processes, monitors checking network traffic, and dedicated machines that are passed suspicious software for analysis and which alert the entire system when new malicious code is found.

Exercise 5.1 (Determining Software Emulation) *Determine a way which a virus might be able to decide whether it was being executed on a software CPU emulator so that it could disguise itself, and a counter measure that could be taken by the emulator to detect such a disguise.*

5.2 Securing Code

Besides digital signatures being used to authenticate users and messages, they are also widely used to authenticate code. *Code signing* ensures the recipient of a JAR file (such as a Java application, MIDlet suite, or browser applet), or of a C# assembly that the code did originate from the signer and it has not been tampered with enroute since the code was signed. Code is signed by the sender using the private key of a key pair, and can be verified by a recipient if the recipient possesses a trusted certificate for the corresponding public key.

A JAR file can be signed using the command line `jarsigner` utility (found in the `bin` folder of the Java SDK), specifying the keystore and passwords, an optional name for the signed JAR, the name of the JAR file to sign, and the alias of the private key to use for signing:

```
jarsigner -keypass mykeypw -keystore mykeystore.jks
   -storepass mystorepw -signedjar SignedJARName.jar
   JARName.jar mykeyalias
```

Most IDEs such as Eclipse and the Wireless Toolkit have menu options for facilitating the signing of JAR files, as well as the management of certificates and key pairs. When a signed JAR file is received the `jarsigner` utility can also be used to verify the signature, and optionally display the certificates that formed the certificate chain validating the signature:

```
jarsigner -keystore mykeystore.jks -storepass mystorepw
   -certs -verify SignedJARName.jar
```

Usually browsers and mobile devices have their own trusted certificates that they use to verify the signature of code they obtain, so the verification is performed automatically and the user notified as to whether the signature was verified. It must however be remembered that code signing only guarantees authenticity that the code is indeed from the signer, not that the code itself does not perform any malicious function, so a valid signature should not considered a complete guarantee about the code.

.NET also supports code signing. The command line `sn` (Strong Name) utility can be used to generate a key pair and place the pair in an (insecure) file:

```
sn -k myKeyPairFile.snk
```

Then the public key can be extracted from the file:

```
sn -p myKeyPairFile.snk myPublicKey.snk
```

Once a key pair has been obtained the following attribute is placed in the `AssemblyInfo.cs` class for the project indicating that the assembly should be signed:

```
[assembly:  AssemblyKeyFile(@"myKeyPairFile.snk")]
```

Alternatively, a .NET file such as single compiled C# class can be signed or verified using the

`signtool` utility.

Developers of code often want to protect their code against *reverse engineering*. Both Java and .NET come with command line *disassembler* utilities, `javap` and `ildasm` respectively, which present a human-readable version of any compiled bytecode. However, a *decompiler* can instead be used on compiled code to reveal much more of the structure of the original source code, producing a functionally equivalent source file. Most decompilers can obtain quite a bit of the structure of the source, including the original variable and method names (which are retained by a compiler in compiled classes), and a lot of information about how the methods were programmed, apart from comments which are always removed by a compiler. This can potentially put the intellectual property of a programmer at risk, as recipients of compiled code can themselves determine details of how the programmer wrote the software. To counter attempts at reverse engineering code, a developer can use a *code obfuscator*. An obfustactor transforms a compiled class by scrambling and renaming variable, method and class names, replacing pieces of code with equivalent but more obscure code, in-lining code fragments, and possibly optimizing control structures.

For example, using a common decompiler (called JAD) on the compiled class `XMLSignerVerifier` results in a reasonably close copy of the original source file (see the file `XMLSignerVerifierRE`). Passing the compiled class through an obfuscator (called ProGuard) results in a functionally equivalent class file. Decompiling this obfuscated version exposes less information about the source (see the file `XMLSignerVerifierOB`), in particular the field and inner class names have been lost. However, it is clear from this example that obfuscation does not totally thwart attempts at reverse engineering, although it can make decompiled code (particularly for a large project) more difficult to understand. To fully protect code from attempts at reverse engineering it can be encrypted, which would protect its source from all but the intended recipient.

Whenever bytecode for a Java class is loaded into a virtual machine for execution, or a C# assembly is loaded into the .NET common language runtime, the (Java or .NET) virtual machine first performs *bytecode verification*. The verification involves checking the code to ensure issues such as that all variables have been initialized before they will be used, method signatures (parameter and return types) are correctly used, scope rules (private and protected modifiers) are correctly followed, and memory for the program is correctly accessed. Since a Java or C# compiler has itself already checked the source code for such problems it might seem strange that all compiled code is checked again just before the code gets executed. The reason for this is that a skilled attacker could maliciously modify the code output from a compiler (or use a malicious compiler) and try to arrange for some operation that is unsafe for the Java virtual machine or .NET platform, such as accessing a memory location outside the bounds of the program or causing a runtime stack overflow. Hence, verification prevents a tampered class file from damaging the runtime environment. Note that for Java Standard Edition and .NET all bytecode is automatically verified, whereas for Java Micro Edition it performed as a separate step before being installed on a device so to reduce the burden on the device's virtual machine.

Once compiled code has been checked by a virtual machine and allowed to execute its authorization to perform specific operations while running is controlled by a *security manager*, which uses a *security policy* specifying the types of operations for which the code is granted permission. A security manager is represented in Java by the `SecurityManager` class from the `java.lang` package, or its subclass `RMISecurityManager` which is required if dynamic loading of class files is used for remote objects. By default a Java application has no security manager installed, so that all Java operations are permitted. One can be installed and a security policy file specified either with the following lines of code in the application:

```
System.setProperty("java.security.policy","MyApp.policy");
System.setSecurityManager(new SecurityManager());
```

or else when the application is executed:

```
java -Djava.security.manager
   -Djava.security.policy=MyApp.policy MyApplication
```

Besides a specified policy file there are standard locations where a security manager checks for further policy files, the `java.policy` file in the `lib/security` folder of the Java runtime, or else in a file called `.java.policy` in the user's home directory (standard locations can be

specified in the `java.security` properties file). A security manager adds (the union of) all the permissions granted in any of these policy files, unless `==` is used in the command line in place of `=`, which instructs the security manager to only use the specified policy file. A system administrator can modify the `java.security` properties file to specify security policy files that cannot be edited by the user and that additional policy files are not allowed to be specified by the user. For comparison, .NET instead uses four policy files, *enterprise* (used by a system administrator), *machine* (used by a machine administrator), *user* (which can be modified by the user), and *application domain* (for a particular application), and takes the intersection of all the permissions granted. Hence, an operation is only permitted in .NET if permission for it is granted in all the policy files. These policy files are modified through the .NET administrative tools.

Java and .NET each include permissions for accessing file systems, network communication, the display, reflection (the ability to determine and create instances of a class given an instance), accessing the system clipboard, controlling threads, particular types of database access, and using a printer. In Java the location of a key store (holding certificates) can be given at the start of a security policy file before the first `grant`:

```
keystore "file:mytruststore.jks", "JKS"
grant signedBy alias codeBase uri Principal PrinClass name
{  permission SecurityPermissionClass Action;
   ⋮
   permission SecurityPermissionClass Action;
};
⋮
```

A `java.io.FilePermission` specifies a file target:

file for the specified file,
`*` for all files in the current directory,
`-` for all files in the current directory or in any subdirectory of it,
directory`/` for the specified directory,
directory`/*` for all files in specified directory,
directory`/-` for all files in specified directory, or in a subdirectory
`<<ALL FILES>>` for all files in the file system,

followed by a comma and any combination of `read`, `write`, `execute`, `delete`. A `java.net.SocketPermission` specifies a host:

hostname for the specified host or IP address,
`localhost` or `""` for the local host,
`*.`*domainSuffix* for any host that ends with the given suffix,
`*` for all hosts,

followed by a port range:

`:`*n* for just the specified single port,
`:`*n*`-` for all ports numbered *n* and above,
`:-`*n* for all ports numbered *n* and below,
`:`*n1*`-`*n2* for the specified range of ports,

followed by a comma and any combination of `read`, `write`. Custom security permission classes can be made by extending the `Permission` class or one of its subclasses from the `java.security` package, providing implementations for the methods `getActions`, `equals`, `hashCode`, `implies`.

Applets that are loaded over a network by a browser use a security manager provided by the browser and are usually assigned a very restricted security policy. By default the security policy prevents applets from performing operations which might put the client at risk, such as reading and writing files on the client file system, and from making network connections to any host other than the host from where the applet was obtained. In addition, applets loaded over a network are prevented from starting other programs on the client, they are not allowed to load libraries, or to define native method calls (to prevent applets from obtaining direct access to the computer). Applets that need some of these permissions granted are typically signed and

a security policy file is provided to the browser.

Exercise 5.2 (Granting Permission to Signed Code) *Prepare a GUI that contains a text field for entering the name of a file and a text area for displaying the contents of the file (ensuring the file reader catches a possible* `SecurityException`*). Then sign the GUI and create a suitable security permission file so that the GUI can only read from the file system if it is signed using a key for which there is a trusted certificate.*

5.3 Firewalls

A *firewall* is hardware and/or software technology that is placed on the perimeter of a network and which acts as a security barrier around the network, examining and filtering all communication passing in and out of the network. Firewalls are used to restrict the Internet services available to users and help protect machines within the network from external attacks. Filtering rules can restrict communication to and from certain IP addresses or restrict traffic to only certain protocols as determined by the firewall administrator. A firewall can be classified as follows:

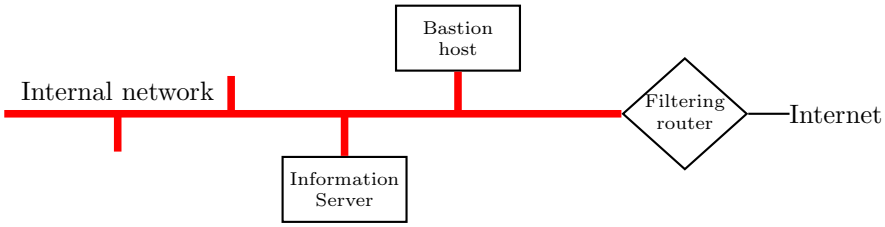
Packet filtering firewall which filters packets at the network protocol level, checking each IP packet against a list of security criteria, such as source and destination IP addresses, port numbers, and the communication protocol. Packets that do not meet the criteria are either discarded or else result in a message back to the sender. Packet filtering is usually performed by a router, providing some basic but limited security without adversely affecting the performance of the network.

Circuit level gateway which filters packets at the transport or session protocol layer, monitoring TCP handshaking to determine whether a requested TCP session is legitimate, such as ensuring that was initiated by a known IP address within the network, not externally. Rather than permitting a direct end-to-end TCP connection between a machine inside the network with one outside, a circuit level gateway instead sets up its own connection with either end and relays TCP segments between the two connections, performing *network address translation* (NAT) between the private IP addresses within the network and temporarily assigned IP addresses. Any machine outside the network thus communicates with the gateway, so actual IP addresses of machines within the network are hidden.

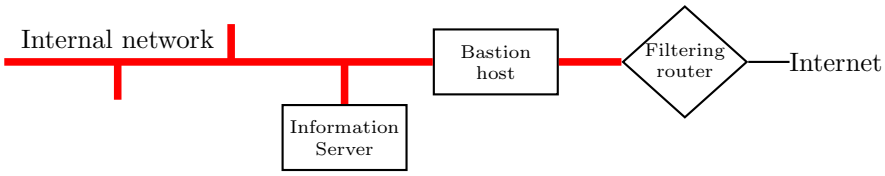
Application level gateway (proxy) which filters packets at the application protocol layer, only allowing application protocols for which there is a corresponding proxy in the firewall, such as the freely available Squid web proxy which only allows HTTP traffic and can inspect `GET` and `POST` requests. Application level proxies offer a high level of security as they can inspect the data being transmitted but have a significant impact on network performance.

Stateful multilayer inspection firewall which combines aspects of the other three types of firewall, filtering packets at network layer, session layer, and using algorithms to process application layer data rather then using application-specific proxies. A stateful firewall holds attributes of each connection in a table so that packets for connections that have already been screened can be quickly processed (interestingly, the most common DDOS attack is currently a SYN flood which attempts to overflow this firewall table). It offers good security and performance but is more complex to administer.

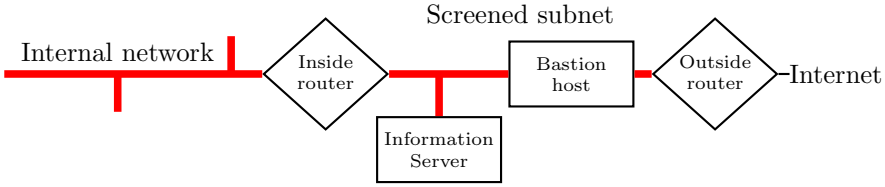
In the *screened host with single-homed bastion* firewall configuration a packet filtering router is used to block all packets except those that are to or from a secure *bastion host* which performs circuit level or application level filtering. Optionally the router might also allow communication with some public information servers (such as web servers).



However, if the router were to be compromised by an attacker, then the machines on the internal network would become vulnerable as packets could then be sent directly to them. To counter this risk, the *screened host with dual-homed bastion* firewall configuration places the bastion host between the internal network and the router, providing dual layer security, requiring all communication to physically pass through both the router and the bastion host.



The *screened subnet* firewall configuration provides a third security layer by placing another packet filtering router between the public (screen subnet) part of the network (consisting of the bastion host, public information servers, modems) and the internal network.



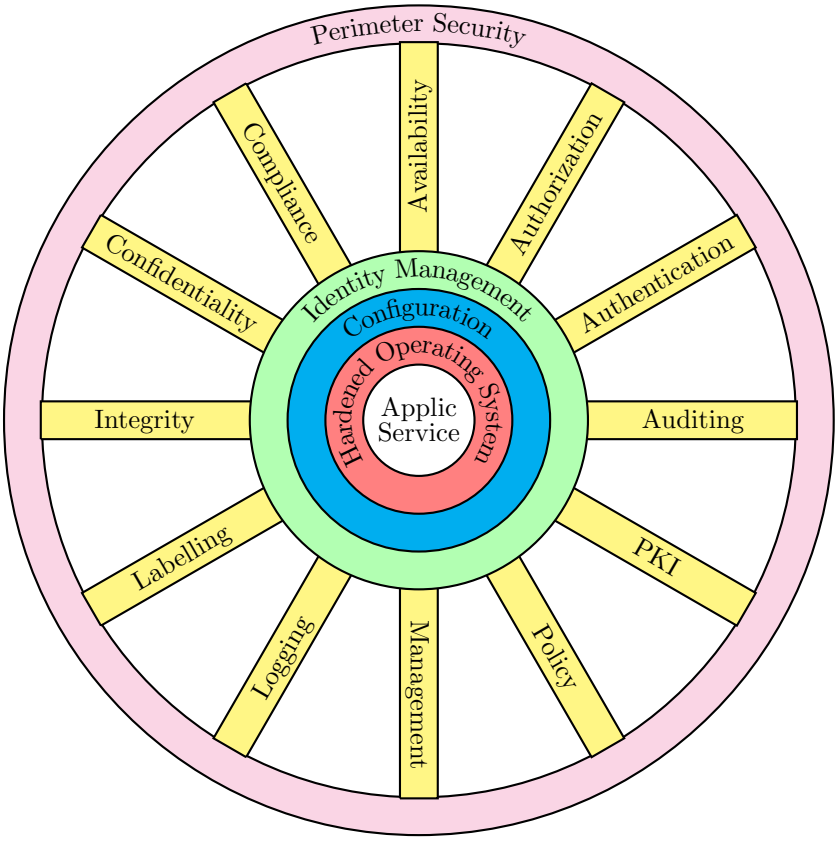
The outside router only allows external access to the screened subnet, and does not allow direct access to the internal network. Similarly, the inside router only allows the internal network access to the screened subnet.

The *Sockets protocol* (SOCKS) is an intermediate protocol between the transport and the application protocol layers that enables a client located behind a firewall to transparently communicate with an external server. When a TCP or UDP client wants to communicate with a server located outside of the network firewall a TCP connection is made to a SOCKS proxy server (running by default on port 1080), the client authenticates itself with the server, and if authenticated it sends the SOCKS server a *relay request* for performing TCP or UDP communication with a stated external server. A SOCKS proxy can be specified in Java through the system properties `socksProxyHost` and `socksProxyPort`.

If a firewall does not allow application data to be communicated directly using a protocol such as TCP then an HTTP tunnel can be set up. An *HTTP tunnel* wraps application data inside HTTP packets which are usually allowed to pass through most firewalls. The HTTP packets are then processed by a *mediator server* outside the firewall which converts the data back to the original application data and forwards it to the intended server. The mediator also wraps the response from the server and sends it back as an HTTP response to the application.

As an application of tunneling through a firewall, note that the class `SSLEchoClient` from Section 4.3 cannot itself connect to an external SSL host from behind a firewall. However, the example class `SSLTunnelSocketFactory` can be used as its `SSLSocketFactory` to create an `SSLSocket` for accessing an SSL host via a proxy tunnel through the firewall. First, a TCP connection is made to the proxy server using a normal `Socket` and a `CONNECT` is sent to the proxy, followed by the name of the SSL host, a colon, and the port of the SSL host. A `CONNECT` method is an HTTP method that requests the proxy to just forward the communication to the specified host and pass the response back. If the proxy agrees to the request (possibly requiring authentication of the client via a Base 64 encoded user name and password) then an `SSLSocket` is layered on top of the `Socket`. Hence the proxy will be unable to inspect the communication it is forwarding between the client and the SSL host.

A *security wheel* (taken from the book *Core Security Patterns* by Steel, Nagappan, Lai) illustrates all the security components for a secure system. At the hub of the wheel is the business logic of the application or service that must have security incorporated into its design. Surrounding this is a hardened operating system (an operating system that has its own security with unnecessary features removed), which is securely configured to provide reliable provisioning mechanisms, and which is protected by an identity management system. The spokes of the security wheel represent the 12 core security services that are incorporated:



auditing which provides regular records about the application or service activity, to support forensic investigations and regulatory compliance,

authentication which verifies the identity of a subject,

authorization which determines whether a subject is permitted to access the various features provided by the application or service,

availability which ensures reliable and timely access to the application or service,

compliance which ensures that standards or regulatory requirements are met,

confidentiality which ensures information is safe from unauthorized access during transmission and storage,

integrity which ensures information is not tampered with by unauthorized subjects,

labelling which appropriately classifies information according to its security level to prevent unauthorized disclosure,

logging which provides records of events for the diagnosis of problems,

management which provides mechanisms for the central administration of security operations,

policy which provides rules and procedures for access control,

public key infrastructure which provides key management and distribution facilities.

All the spokes must be in place to ensure a robust security architecture. Beyond the spokes is the perimeter security, consisting of a firewall along with intrusion prevention and detection systems. Note that it is essential when creating a secure system that all these security components are considered throughout the development process rather than attempts made to add security after the application or service has been developed.

Exercise 5.3 (Proxy Tunneling Through a Firewall) *Implement a simple firewall that allows proxy tunneling to an external SSL host and test it using a modified* `SSLEchoClient`.

SSL TUNNELING

```
/**
   A factory class that can be used to create SSL sockets which use tunneling of SSL
       through a proxy. Adapted from JSSE samples
   @author Andrew Ensor
*/
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import javax.net.ssl.HandshakeCompletedEvent;
import javax.net.ssl.HandshakeCompletedListener;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import sun.misc.BASE64Encoder; //undocumented Sun&IBM VM utility class

public class SSLTunnelSocketFactory extends SSLSocketFactory
{
   private String tunnelHost;
   private int tunnelPort;
   private String tunnelUserName;
   private String tunnelPassword;//should not store password as String
   private SSLSocketFactory defaultFactory;
   private boolean socketConnected;

   public SSLTunnelSocketFactory(String tunnelHost, int tunnelPort,
      String tunnelUserName, String tunnelPassword)
   { this.tunnelHost = tunnelHost;
      this.tunnelPort = tunnelPort;
      this.tunnelUserName = tunnelUserName;
      this.tunnelPassword = tunnelPassword;
      defaultFactory = (SSLSocketFactory)SSLSocketFactory.getDefault();
      socketConnected = false;
   }

   public SSLTunnelSocketFactory(String tunnelHost, int tunnelPort)
   { this(tunnelHost, tunnelPort, null, null);
   }

   public String[] getSupportedCipherSuites()
   { return defaultFactory.getSupportedCipherSuites();
   }
   public String[] getDefaultCipherSuites()
   { return defaultFactory.getDefaultCipherSuites();
   }

   public synchronized boolean isSocketConnected()
   { return socketConnected;
   }

   public synchronized void setSocketConnected(boolean socketConnected)
   { this.socketConnected = socketConnected;
   }

   public Socket createSocket(Socket s, String host, int port,
      boolean autoClose) throws IOException, UnknownHostException
   { // create a regular socket to tunnel through the proxy host
```

```
   Socket tunnel = new Socket(tunnelHost, tunnelPort);
   doTunnelHandshake(tunnel, host, port);
   // layer an SSL socket over the top of the regular socket
   SSLSocket sslSocket = (SSLSocket)
      defaultFactory.createSocket(tunnel, host, port, autoClose);
   sslSocket.addHandshakeCompletedListener(
      new HandshakeCompletedListener()
      { public void handshakeCompleted(HandshakeCompletedEvent e)
         { System.out.println("Handshake completed with peer host "
               + e.getSession().getPeerHost()
               + " and assigned session ID " + e.getSession());
            setSocketConnected(true);
         }
      });
   return sslSocket;
}

private void doTunnelHandshake(Socket tunnel, String host, int port)
   throws IOException
{ OutputStream os = tunnel.getOutputStream();
   String connectionString = "CONNECT " + host + ":" + port
      + " HTTP/1.0\n" + "User-Agent: "
      + sun.net.www.protocol.http.HttpURLConnection.userAgent
      + "\r\n";
   if (tunnelUserName!=null && tunnelPassword!=null)
   { // add the Base 64 encoded user name and password
      BASE64Encoder encoder = new BASE64Encoder();
      String encodedString = encoder.encode((tunnelUserName + ":"
         + tunnelPassword).getBytes());
      connectionString += "Proxy-Authorization: Basic "
         + encodedString + "\r\n";
   }
   connectionString += "Content-Length: 0\r\n"
      + "Pragma: no-cache\r\n\r\n";
   // send the connection HTTP request using ASCII7 encoding
   byte[] connectionBytes;
   try
   { connectionBytes = connectionString.getBytes("ASCII7");
   }
   catch (UnsupportedEncodingException e)
   { connectionBytes = connectionString.getBytes();
   }
   os.write(connectionBytes);
   os.flush();
   // obtain the connection reply
   InputStream is = tunnel.getInputStream();
   List<Byte> responseList=new ArrayList<Byte>(); //response header
   int newLinesSeen = 0;
   boolean headerDone = false;
   while (newLinesSeen < 2)
   { int data = is.read();
      if (data < 0)
         throw new IOException("Unexpected end of response");
      if (data == '\n')
      { headerDone = true;
         newLinesSeen++;
      }
      else if (data != '\r')
      { newLinesSeen = 0;
         if (!headerDone)
            responseList.add((byte)data);
      }
   }
```

```
      // convert the HTTP response into a string using ASCII7 encoding
      int responseLength = responseList.size();
      byte[] responseBytes = new byte[responseLength];
      for (int i=0; i<responseLength; i++)
         responseBytes[i] = responseList.get(i);
      String responseString;
      try
      { responseString = new String(responseBytes, 0, responseLength,
           "ASCII7");
      }
      catch (UnsupportedEncodingException e)
      { responseString = new String(responseBytes, 0, responseLength);
      }
      // check that response was successful
      if (responseString.toLowerCase().indexOf
         ("200 connection established")<0)
         throw new IOException("Unable to tunnel through "+tunnelHost
            + ":" + tunnelPort + ". Proxy returned " + responseString);
   }
   // overridden method of SocketFactory
   public Socket createSocket(InetAddress host, int port)
      throws IOException, UnknownHostException
   { return createSocket(null, host.getHostName(), port, true);
   }

   // overridden method of SocketFactory
   public Socket createSocket(InetAddress address, int port,
      InetAddress localAddress, int localPort)
      throws IOException, UnknownHostException
   { return createSocket(null, address.getHostName(), port, true);
   }

   // overridden method of SocketFactory
   public Socket createSocket(String host, int port)
      throws IOException, UnknownHostException
   { return createSocket(null, host, port, true);
   }

   // overridden method of SocketFactory
   public Socket createSocket(String host, int port,
      InetAddress localHost, int localPort)
      throws IOException, UnknownHostException
   { return createSocket(null, host, port, true);
   }
}
```

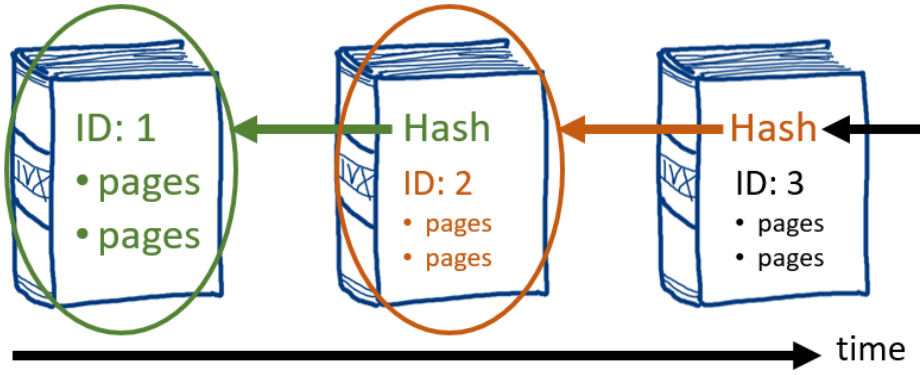
Chapter 6

Blockchain & Cryptocurrency

The *crypto* in cryptocurrency comes from cryptography; thus, a cryptocurrency is a currency that is cryptographically secured. In 2017 the term *cryptocurrency* was added to the Oxford English Dictionary along with *bitcoin* and *blockchain*. This chapter will introduce the technology that enables bitcoin and some of the security implications of a blockchain based application. Blockchain and bitcoin have become synonymous but there are important differences. First we will look at the blockchain as a data structure.

6.1 Blockchain Data Structure

A blockchain is a data structure whereby a single block of data contains a reference to a previous block. Usually this has occurred in the past, and so a chain of blocks can represent a chronological ordering of data. When a new block is created it must include a reference pointer to the previous block in the chain. This is done by including a hash of the previous block.

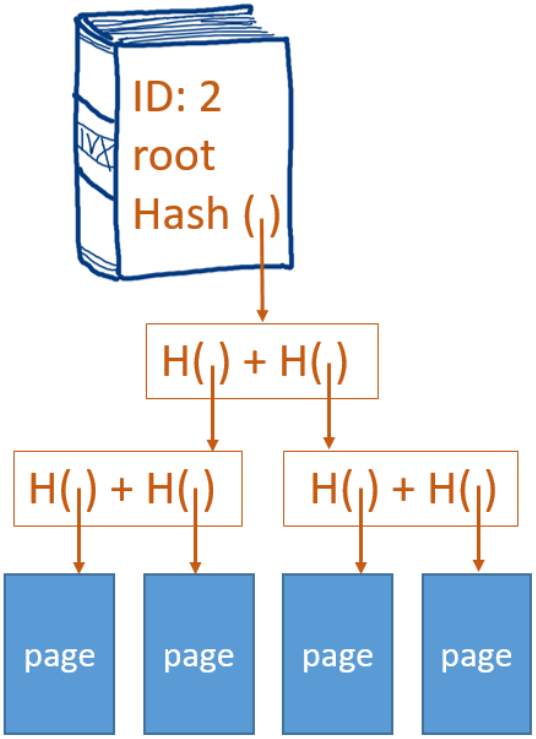


The blockchain must be created one block at a time and mass deletion or appending of new blocks is not possible to maintain the correct hash linking. If there are multiple new blocks to be added to the data structure, they must queue and be processed according to some predetermined rules. In this sense, a blockchain is considered to be immutable, and append-only. The first block, labelled ID:1 above, is often called the genesis block, and this is the only block that has been created by design at the inception of the program. All other blocks are added in sequence according to the protocol of the system.

Recall that a hash function takes a variable length input and produces a fixed length output with no discernible pattern. A good hash function is one-way which means that it is easy to calculate (verify) but difficult to reverse. Given a hash it should be computationally infeasible to both produce the data that created the hash, and to find a message that produces the same hash. This scenario is called a collision and for the blockchain to be secure it should be composed using a collision-resistant hash function.

If an adversary wishes to modify data in a block – such as financial transaction data – the resulting hash pointer will need to be updated. (Even a variation of a single bit yields a

different hash.) So rather than having to store the entire data set for verification, we can store the hash pointers and easily see if they are correct. Our adversary that altered some data would have to change every subsequent hash pointer to the tip of the blockchain. Simply storing the most recent hash in a place that can't be modified is enough to verify the whole chain. This can be done by keeping multiple copies in different locations.



The pages inside the books (data) can also be represented using hash pointers through a *Merkle tree*, named after computer scientist and cryptographer Ralph Merkle. The leaves of the binary tree structure are the data and a hash pointer is created for every pair of leaves. When there are two pairs of leaves, the two hash pointers are combined and hashed into a third hash pointer. This continues until the root of the tree is a single hash representing all the data. The Merkle tree is secure in the same way as the blockchain – if Eve tries to modify some data, then the root hash pointer will change.

A blockchain data structure can be *permissioned* and require some authentication to browse elements and append blocks, or *permissionless* allowing anyone to view the data and perhaps add blocks depending on the rules of the protocol. There are applications for both styles of blockchain and, as many believe, currency works best as a public and transparent data structure.

6.2 Cryptocurrencies

Financial transaction data[1] is an ideal use case for a blockchain data structure as it is generally chronological, and does not contain any cyclic elements (such as changing your name, which was appended to the database at birth). Hash pointers only work in a system like this because if a previous element changes, the subsequent hash pointers would need to be updated. Transactions themselves may be reversed, but from an accounting perspective this is simply a debit back to the customer represented as a second transaction. Both transactions are stored and should be retrievable in the future.

A monetary system is an evolution of social trust. In small tribes or individual families, money

[1]Blockchains are sometimes called open ledgers in reference to an accounting ledger that tracks income and disbursements; similarly, distributed ledger technology refers to a blockchain combined with distributed computing architecture.

is not necessary because the members trust each other to keep a tally[2] of various debts and credits. In the case of a cryptocurrency, hash functions and public key cryptography allow a user to trust a stranger with their money. A monetary system can be constructed with a number of axioms.

Any monetary system must be:

Fungible No single token is differentiable in value from another. Artwork exchange would be non-fungible because of the inherent personal differentiation of value.

Durable The tokens must last at least long enough to be used in two transactions: one to receive them and one to spend them. This is solved easily with digital tokens.

Scarce If everyone had easy access to it, it would not be valuable. This is a central problem with digital currency because copying digital entities results in a perfect duplicate indistinguishable from the original.

Divisible Bartering with whole goods is difficult because value might not be aligned. This is a benefit of a digital system; coins can have a very high resolution, much greater than $0.01.

A cryptographically secure financial transaction system should embody the following additional elements:

Decentralization Under control of the users; as opposed to the administrators. This helps ensure that value cannot be stolen, diluted, or written off by powerful actors in the system.

Uniqueness Coins should only be allowed to be spent once. This is a key barrier to overcome when transacting in a digital medium.

Predictability of Supply Creation and distribution of new tokens according to rules the users agree with.

Distributed Network The network should be invulnerable to attack or regulatory control from a single entity.

Security Ownership of addresses (coins) and the ledger should be computationally secure and hackers or bad actors should not be able to modify the historical record.

A Brief History: Before Bitcoin

In 1983 David Chaum came up with a scheme for electronic cash involving distribution, tracking, and spending of coins that relied on a bank entity for clearing. Chaum's idea was to use what are called *blind signatures* to keep the identity of the user hidden from the central clearing house, while still being able to trust that the hidden user had viable funds to spend. If the bank issues a new digital note to you and lets *you* pick the serial number and then the bank signs it, there is now a new note in existence guaranteed by the bank but they don't know who they gave it to. If you tried to spend your new digital note twice the bank could step in and compare the transactions and call you out on the double-spend attempt.

Chaum started DigiCash in 1989 to allow users to conduct anonymous online transactions. DigiCash and others in the intervening years relied on banks for the issuance of new currency, and although had unique privacy features, banks largely did not need the new technology and did not adopt it. Much of the work since this time has involved trying to remove the bank from the equation to create a decentralized currency.

Another issue to consider is the minting of the digital coins. A good solution requires a cost structure similar to the printing of modern notes where capital expenditure is large to acquire printing presses and develop the security features. This cost goes down over time as subsequent print runs cost less and less making it difficult for single users to print their own money. For digital minting, the computational power required to copy a digital serial number is negligent and so a more intensive process is required.

In 1997 original cypherpunk Adam Back came up with *Hashcash*. Hashcash was developed to solve the problem of email spam. DDoS attacks on email servers can be mitigated by hashing a proof of computational effort and including it in the email header. Spammers and phishers must

[2]The term *tally* comes from an old British system of keeping debts with notched wooden sticks called tallies.

send thousands of emails to get a response and so a small delay in each mail sent can be very frustrating. The delay is negligible for the average user as their own processor could compute the proof-of-work during composing their message and not notice the cpu time required. The work that the cpu is doing is finding a solution to a hash puzzle. A variant of Hashcash is used in bitcoin and offers a solution to the coin minting problem. Section 6.4 discusses proof-of-work.

6.3 Bitcoin

In 2008 a whitepaper emerged by Satoshi Nakamoto that combined many of the elements we have studied thus far and solved the double-spend problem in a distributed system. The new feature in bitcoin is called *emergent consensus*[3] and allows a network of independent users and operators to monitor transactions against a global ledger. Every node independently verifies transactions and the network globally agrees on new blocks to be appended to the blockchain. This agreement of the blockchain state by its users is the last piece in the digital cash puzzle thereby eliminating the need for a trusted intermediary. Bitcoin tied together the following modules to create a distributed, cryptographically secure financial system:

Decentralization Bitcoin software is open-source so anyone can run the software and set up a node in the network. There has been considerable growth since 2009; estimates indicate there are around 10,000 full nodes online. Each node contains a copy of the blockchain and can independently validate transactions. This eliminates single-point-of-failure problems common in centralized systems.

Consensus All participants must agree on the state of the ledger. It is this trust in the ledger that allows one party to interact with another that they did not previously know or trust personally. Sometimes this is referred to as *trustless*.

Uniqueness Every transaction has a hash pointer within a block. In this sense all the transactions are unique. Individual coins do not actually exist, rather hash pointers can indicate unspent transaction outputs that can show `9.87654321 bitcoin`[4], for example. Whoever has the private key to these unspent outputs is considered the owner.

Supply Stability Every block that is added to the chain results in the minting of new bitcoins. This amount is halved about every four years so the total supply is a decreasing geometric series. The maximum number of bitcoins that will be minted is just under 21 million.

Security Bitcoin uses the NIST specification `secp256k1` elliptic curve digital signature algorithm. This keeps users private keys secure and authenticates transactions in the network.

How do you keep the state of the whole blockchain safe from sticky fingers? What if one party decides to spend their coins to buy some gold, and then after receiving the gold, they spend their coins *again*, this time sending them to another address they control. We need a way to distinguish between the two transactions without mediation. The next section will discuss this issue: how can everyone agree on what transaction is valid?

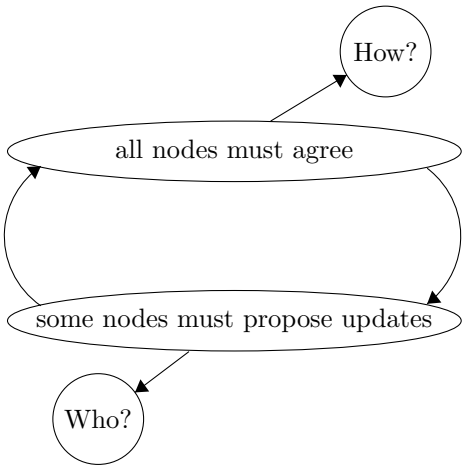
6.4 Consensus

Solving the situation described above, known as the *double spend problem*, is the final piece of the puzzle that was put in place for trustless decentralized digital cash to succeed. First we will describe the *Byzantine Generals Problem*, a famous computer science communication problem.

Imagine three generals of the same army, all wanting to attack an enemy fortress at dawn the next morning. Each general knows their individual divisions cannot win the enemy fortress. Each general also knows that with the help of the other's forces they can win the position. So General *Alpha* sends a messenger to General *Beta* with the message "We attack at dawn." This is where the problems start. There are many scenarios that could play out. The messenger could be captured by the enemy; the messenger could be a double-agent; the messenger could get lost and not deliver the message; General Beta could double-cross Alpha, etc. To win the battle, the three generals must come to consensus and agree to attack at dawn. Practically

[3] The term Nakamoto consensus is gaining popularity in the literature.
[4] A bitcoin is divisible into 8 decimal places; 0.000 000 01 is called a satoshi.

speaking, each general must receive confirmation from each of the other generals that they received the message *and* are in agreement. This scenario maps nicely onto cryptography and message passing.

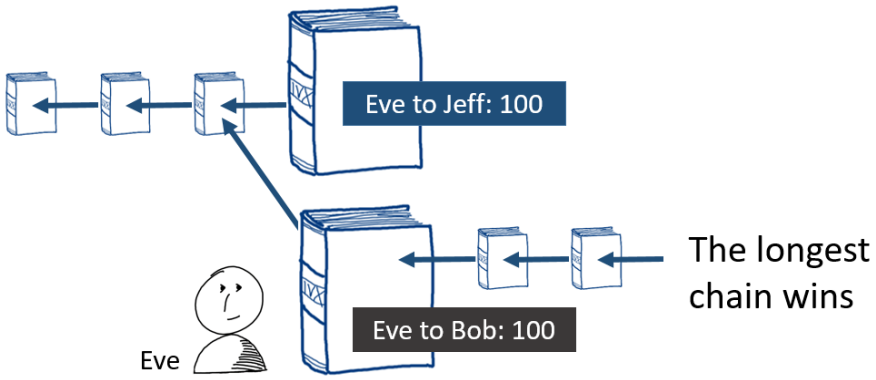


A decentralized blockchain application has many nodes in the network and each of them has a copy of the blockchain. If Bob wants to send Alice his tokens, the nodes in the network must first agree that Bob has available tokens to spend, and then update the state before Bob can double-spend his tokens. The network needs to be in a state of *consensus* for people to trust it. Consensus in the Byzantine Generals problem has been proven to be impossible if more than a third of the nodes are malicious.[5]

Bitcoin does not exactly mirror the model of the Byzantine Generals and so is not impossible to reach consensus. In other words, bitcoin does achieve consensus despite Byzantine behaviour. More practically, bitcoin achieves a state of *emergent* consensus. This means that over time as blocks get added to the chain, a general state emerges that everyone agrees on. So how does this occur without clear rules? A closer look at the double-spend attack will illustrate how nodes come to agreement.

The Double-Spend Attack

A dodgy actor, Eve, could manipulate the blockchain by sending some coins to Jeff and receiving a good or service. Then she could send those same coins to Bob's address whom she also controls. If the transaction involving Bob's address is validated on the blockchain, then Eve would have successfully double-spent the same coins. She now has the coins in Bob's address and the goods from the transaction with Jeff.

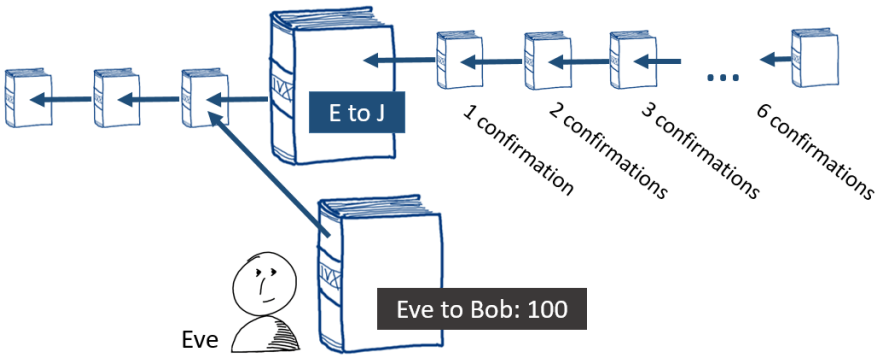


How can Jeff avoid this scenario, especially when he does not know the other party he is transacting with? The solution is to wait. Over time as more blocks are added, say every ten minutes, the older blocks are more likely to be valid.[6] Stated another way, as more blocks

[5]The term Byzantine in computer science indicates an unpredictable actor–this includes malicious activity.

[6]Standard confirmation time for bitcoin is six blocks, or one hour. The number of blocks to wait until considering a transaction to be confirmed is a personal preference.

are added to the chain there is an exponential decrease in the probability of a block being rejected by other nodes. This is the method that secures the blockchain against malicious nodes attempting to double spend.



And what about Eve changing history? She would have to alter the block where the transaction is stored, as well as any subsequent blocks. The older the block is, the more computational effort is required because the hash pointers are linked. Consider that Eve is also competing against other nodes in the network that are honest. (We will return to the case of dishonest nodes when discussing a 51% attack.) The next question is how to keep nodes honest, especially when dealing through the medium of currency?

Proof-of-Work

Here is an important distinction where bitcoin varies from other methods of reaching consensus. There are incentives for nodes to act honestly that are built into the protocol. The first is called the *coinbase* transaction and awards freshly minted bitcoins to whoever added the block to the chain. This is how new bitcoins come into circulation. The second is from transactions fees. By listening to the network, validating transactions, and including them in a block, whoever is operating the node can choose to include transactions that also offer an extra fee. Because bitcoin itself is designed to be digital money, this makes perfect sense and is why cryptocurrency is considered the killer app for blockchain.

Referring back to the state diagram, who gets to propose these blocks and earn the block reward? How do we ensure the distribution is random? How are Sybil attacks prevented? Mining is the answer and provides the incentive structure to maintain this delicate balance.

Mining

A miner is a network participant that contributes their computing power in a demonstrable way. A fair way to allocate the incentives would be by some resource that can not be gamed or monopolized. One such way is by computing power, and is called *proof-of-work*. Another method is by value held in reserve and is called proof-of-stake. Proofing methods is a very active research topic as it has many game theoretic variables.[7] Bitcoin miners participate by using their computing power to validate transactions and suggest new blocks. For this effort they receive rewards in proportion of their contribution to the network as a whole. Bitcoin uses a `SHA256` hashing algorithm as the hash puzzle that miners have to find a solution for.

```
Block = Hash(nonce||previousHash||data||data||...||data) < target
```

Hash function output has a random distribution and so to find a block, your hash must be below a certain target level. The target level will be the hash size with a certain number of leading zero bits, for example:

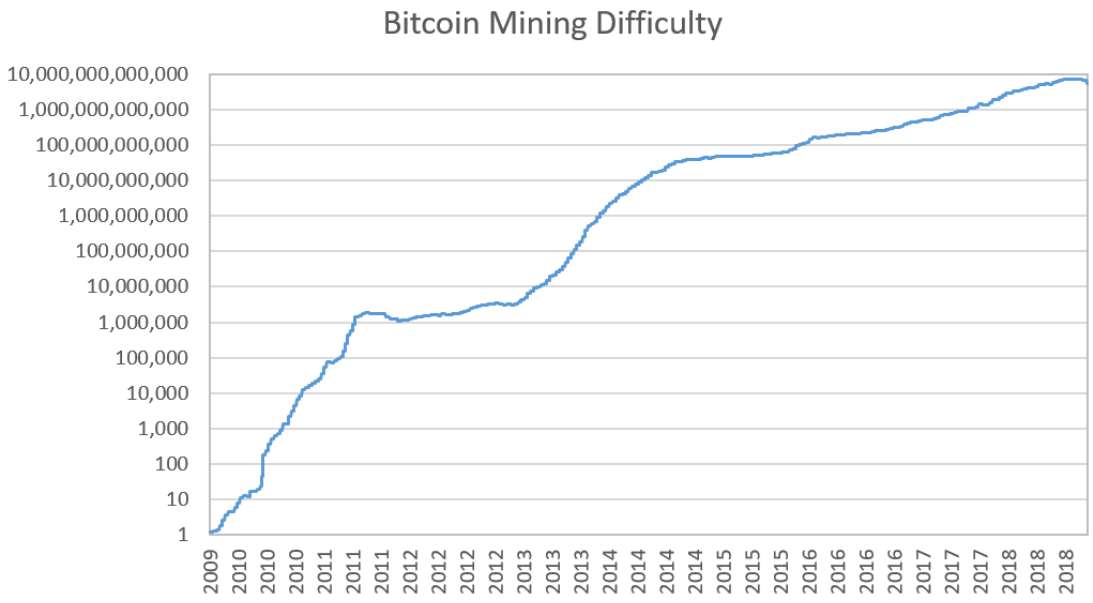
```
0000000000000000000117c467ab5336077cb04f7f70ea6ebcd68e0b3ef6cf909
```

was the successful hash of block `529283`. The only way to find a hash with a smaller value than the target is to change a nonce value and re-hash the bundle of transactions over and over. When a target is hit, the block is broadcast to the network as a proof of computational work done in solving the hash puzzle. The bits here should be randomly distributed, like a lottery, to

[7] Other methods include DPoS: delegated proof-of-stake, PoET: proof of elapsed time, and PoA: proof of authority.

prevent gaming the system and earning more rewards than your proportion[8]. Computing cycles in the bitcoin network is called *hashpower* in reference to `SHA256`. As more miners come online, the total hashpower increases leading to greater overall probability of successfully hashing a value below the target. To keep the temporal distribution of blocks even, this target difficulty automatically adjusts according to the protocol every 2016 blocks, or approximately two weeks.

Below is a log plot of the mining difficulty from 2009-2018 showing that the difficulty has increased exponentially. The slope roughly correlates to the growth of the network.[9]

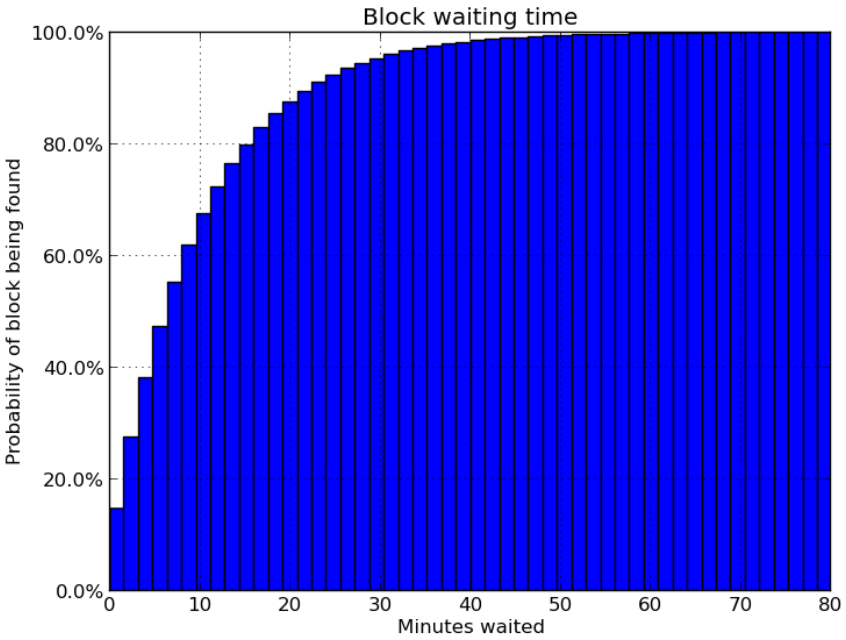


Block Time

The difficulty adjustment aims to keep the time between blocks (successful hashes) at around ten minutes. In the early days Satoshi and a few others could use their PC processors to find blocks every ten minutes. The hashpower has steadily increased and so has the difficulty target to keep the block time constant. Finding a hash of a block that is below the target size is a discrete event; it is either below or it is not. As with a lottery, it is only a matter of time before a hash is found, and the previous hash is independent of the current attempt. Statistically, this is a *Poisson* distribution.

[8] As the bitcoin network has matured, dedicated hardware called ASICs (application specific integrated circuits) to solve the `SHA256` algorithm have dominated. It is no longer feasible for a single participant to mine bitcoin without dedicated hardware.

[9] Source: https://www.blockchain.com/charts/difficulty



From the graph[10] you can see its possible to find a block immediately after the last one, but unlikely. Similarly, its possible that no block could be found for an hour or two, but unlikely. In practice, the average block time is just under ten minutes because as more miners join the network it is easier overall to find blocks, until the difficultly is reset.

Consensus Summary

In summary, consensus of nodes in the network emerges from an implicit agreement on the longest chain of blocks. The nodes are all agreeing that this blockchain represents the most proof of computational work. Miners will abandon shorter chains to compete to build on the longer one to win the block reward. The transactions in this chain will have an increasing probability of being accepted over time as new blocks are mined.

6.5 Security

The first security consideration was the double-spend attack as discussed above. The next weakness comes at the network level. To this point we have said that most nodes in the network will act honestly and the mining block reward incentivises this behaviour. If a collective forms with a considerable portion of nodes this could lead to a 51% attack.

51% Attack

Should a single entity gain control of more than half of the hashpower in the network, this could lead to a 51% attack. The attacker can't steal coins directly as this involves subverting elliptic curve cryptography. The attacker can however user their majority status in some interesting ways. At $> 50\%$ you have the ability to find more blocks and direct consensus of the blockchain. The attacker can censor transactions by refusing to add blocks containing someone's address. This would be akin to blacklisting certain addresses, but does not completely exclude these from being included by honest nodes (the 49%). Next, you may say okay, lets just increase the block reward and award myself 1 million coins for the next block found. This will be rejected by the network because the block reward is hard-coded and so the malicious "plus-one-million" transaction can never be spent.

An adversary doesn't necessarily need 51% of the hashpower, but with a large number of nodes in the network they could facilitate a double spend attack by the following steps.

[10]Source: https://en.bitcoin.it/wiki/Confirmation

1. Broadcast a transaction they intend to double spend that the receiver believes is legitimate.
2. Mine a different branch of the blockchain that doesn't include their transaction from (1), but does include a transaction to themselves.
3. Continue mining the forked branch until the illegitimate transaction is accepted by the receiver and then broadcast their alternate blockchain.
4. As consensus relies on the longest chain representing the most proof of work, this new (and falsified) chain will now be considered valid.

The main practical threat to a single entity controlling majority of the hashpower is that it will lead to centralization and cause users to abandon the system altogether. Once hashpower gets close to this threshold it is also possible that developers will step in and update the software to limit this behaviour. It is unknown how this will play out in the future. Thus far the bitcoin network has been relatively decentralized, but a few mega mining companies have emerged. There have been some cases of 51% attacks on alternative cryptocurrencies such as Verge, Horizen, and Vertcoin. Smaller cryptocurrencies are more vulnerable to a large mining pool shifting their resources for this purpose.

Cryptography

Contrary to popular belief there is no standardized encryption in the bitcoin protocol. As a decentralized system of exchange, there is no need for encryption. All the transactions are stored in the blockchain and accessible to everyone. Access to the private keys controlling addresses is maintained solely through personal security of the user.

So can someone steal my coins? Not by hacking. Elliptical curve cryptography keeps keys safe from cryptanalysis via the difficulty in solving the discrete logarithm problem. The standard curve `secp256k1` was chosen by the designer of the bitcoin protocol. This is a choice unique to bitcoin, as the more common `secp256r1` is used in Transport Layer Security (TLS) for web browsing, email, etc., and discussed in Section 2.4. ECC was chosen because it provides the same level of relative security with smaller keys compared to RSA.

It is imperative that users keep their private keys secret as in any other cryptosystem. There are some added enhancements for encrypting passwords and wallets but these are third party additions and not built in to the protocol.

Key Storage

Storage of cryptocurrency is similar to storage of your online banking details. There are no actual bitcoins to keep safe; there is no `<bitcoin object>` to keep password protected. Remember the blockchain is an open ledger of every transaction in the network. Just like your online banking system keeps track of all account balances and doesn't actually store any fiat[11]. Bitcoin is cryptographically secured, and if you have the private keys to an address, then you control the funds that address references in the blockchain.

A bitcoin address is the public key of a public-private key pair generated by a 256 bit elliptic curve function. The public key is encoded in base-58 to reduce errors in transcription by removing similar characters: `0Oil`.

[11] Fiat is government issued money. New Zealand dollars and Japanese yen are both considered fiat currency.

Cryptocurrencies are more risky than traditional online banking because if you lose your private keys they are not recoverable. There is no password reset, or government issue ID verification to recover your account.

There are a number of way to store your keys:

Hot Wallet This is the most common. A software application that stores your private keys so you can sign transactions, keeps track of your tokens, and maybe offers some additional functionality like multiple addresses, or multiple token support. The risk here is that if you lose your device, your private keys are gone with it. Day-to-day transactions using a smartphone would be done through a hot wallet.

Cold Storage For larger amounts, savings, investments, and custodial services, a cold wallet is recommended. This is a device that is not powered and has no connectivity and stores your private keys internally. When tokens are required, it can be plugged into a USB and connected to the network.

Paper Wallet Once a key pair has been generated and used to receive tokens, that private key will always have ownership of the tokens. The private key can be printed out or written down, sometimes as a QR code for easy scanning. After the memory has been cleared there is no more digital record of the private keys.

Brain Wallet The most hard-core of storage systems is to remember your private keys and destroy all physical and digital evidence of them. Certain memory mnemonics can help with this.

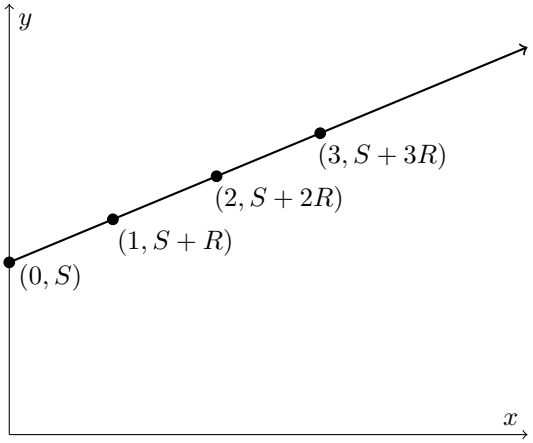
Note that for all these methods including memorization, its possible to *send* tokens to your address without connecting the keys to the network. This transaction will be recorded by all the nodes running the blockchain and propagated accordingly. Private keys are only required to sign transactions (spend tokens). All of these methods are vulnerable to crisis such as fire or theft, and represent a single point of failure. Backing up your keys can solve the crisis issues, but someone could still steal the backup.

Secret Sharing

Lets assume that eventually someone *will* steal our private keys. In this scenario they need the entire key, in the correct order to use it. So why not split the key in two or several parts? This is known as *secret sharing* and works like this: split the key into N shares such that any K of those shares can reconstruct the key, but with $< K$ shares nothing can be learned. Lets split the secret S into 2 shares so that $N = 2$ and $K = 2$. If S is a 128 bit number, a random 128 bit number R is generated and the 2 shares can be R and $S \oplus R$ (bitwise XOR). Either share alone does not help because R is random, and $S \oplus R$ depends on knowing R. In essence the secret has been encrypted with a one-time pad cipher where R is the key. This works for $N = K$. What about for $N > K$?

Say you split the secret in to four shares, give one to each of your trustworthy relatives, and at any time two of them are required to reconstruct the secret. This can be accomplished with

some linear algebra. Any two points can construct a line, but given a third random point it is unlikely to be collinear with the first two. To generate N shares, prepare a line with N points. Now any two of these points can reconstruct the line, and determine the $x-$intercept (S), but any single point is useless because the slope is unknown.



The point $(0, S)$ represents the secret, a large random number less than a large prime, P. The shares are linear combinations modulo P, up to N, where any 2 of them will recover the secret.

$$\begin{aligned} x = 0, &\quad y = S \\ x = 1, &\quad y = (S + R) \mod P \\ x = 2, &\quad y = (S + 2R) \mod P \\ x = 3, &\quad y = (S + 3R) \mod P \\ &\quad \vdots \\ x = N, &\quad y = (S + NR) \mod P \end{aligned}$$

What if you require more than 2 shares necessary to reconstruct the key? If we increase the degree of our share-reconstruction function from linear to parabolic, then we have $K = 3$ is necessary to find S because 3 points can uniquely define a parabola. This can be continued up to $K = N - 1$ shares.

6.6 Privacy

Identity in a decentralized system is based on private keys. Whoever has access to a private key can operate within the network and this activity is directly linked to the key. Just like how digital signatures provide non-repudiation, the activity of an address within an open ledger can be thought of as someone's identity. Keeping with this idea of keys as identities, multiple keys could be multiple identities. You could generate a new identity with a new random key pair and never use the old identity again.

There is a problem here if you want to remain anonymous. Anonymity means that you can still participate in the network but it is difficult or impossible to link your activity with your real identity. Even if one user has multiple addresses, the activities of those addresses is stored in the open – every node in the network has a copy. For this reason, bitcoin is considered to be *pseudonymous*. The activity of one address can be tracked, and this can be mapped to behaviour, possibly providing compelling evidence to link the two. Privacy in cryptocurrencies is a contentious issue and often cited as a founding principle. It goes right back to the core tenet of cryptography – communicating in secret. For a user of cryptocurrency this may be as innocent as purchasing a VPN to use the internet where content and websites may be censored, or for much more nefarious purposes.[12]

[12]For many years bitcoin was synonymous with the *Silk Road*, an online marketplace that remained out of reach of authorities and sold illegal goods. The possibility to purchase illegal material is still a main point of debate against cryptocurrencies. Many people have studied anonymity in markets and cash is still the best way to transact without revealing your identity.

Coin mixing or laundry services aggregate transactions together and wash by sending a large transaction to one or many addresses controlled by the service. Funds are then returned less a service fee. This will help to obfuscate origin and destination of funds from forensic analysis but requires a large amount of trust in the mixing service. This is a weak and impractical solution to transaction anonymity.

Zero-Knowledge Proofs

Privacy can be built in cryptographically at the protocol level using a technique called *zero knowledge*. In cryptography, zero-knowledge proofs are a way to mathematically prove that you know a result without having to reveal the details of how you found it. An example comes from mining. You can prove to someone that you have the hash of a block by revealing the nonce and therefore give away the formula for how you found the hash. A zero-knowledge proof lets you do this without revealing the nonce. The details are non-trivial and out of the scope of the present course. Refer to Foundations of Cryptography by Oded Goldreich, 2001, for an overview.

Several alternative cryptocurrencies have implemented zero-knowledge proofs such as Z-Cash, Horizen, and Ethereum's Byzantium upgrade. These are not compatible with bitcoin as a different set of cryptographic principles had to be coded from the beginning. In Z-Cash, which was derived from zerocoin, the user has the option to transact privately by invoking zero knowledge verification. This takes extra computational resources and reduces efficiency representing a trade-off between privacy and speed.

Code-Breaking

In October, 2018, an image was posted online claiming to contain clues to unlock a wallet holding 310 bitcoin (at the time of writing worth $2.9 million NZD). Just days later the coins were moved from the wallet indicating the private keys had been found in the image. The image below contains a total of four keys, and one with 0.31 bitcoin is still unclaimed. A high resolution image can be downloaded at https://bitcoinchallenge.codes/.