Information Security

Module 205 – Cryptography

Module Notes

2011-2012

Contents

[Section 1 – Introduction and Overview 9](#_Toc298404116)

[1.1 Purpose of these notes 9](#_Toc298404117)

[1.1.1 Using these notes 9](#_Toc298404118)

[1.1.2 Tutorial Exercises 9](#_Toc298404119)

[1.2 Introduction to cryptography 9](#_Toc298404120)

[1.2.1 A Historical Context 9](#_Toc298404121)

[1.2.2 The Electronic Age 10](#_Toc298404122)

[1.2.3 Mathematics 11](#_Toc298404123)

[1.2.4 Summary 11](#_Toc298404124)

[1.2.5 Tutorial Exercise 11](#_Toc298404125)

[Section 2 – Number Theory 12](#_Toc298404126)

[2.1 Modular arithmetic 12](#_Toc298404127)

[2.1.1 The 24-hour clock 12](#_Toc298404128)

[2.1.2 Modulus 12](#_Toc298404129)

[2.1.3 Remainders 13](#_Toc298404130)

[2.1.4 Congruences 13](#_Toc298404131)

[2.1.5 A note on convention 13](#_Toc298404132)

[2.1.6 Modular Relations 14](#_Toc298404133)

[2.1.7 Tutorial Exercises – Modular Arithmetic 14](#_Toc298404134)

[2.2 Exponentiation 14](#_Toc298404135)

[2.2.1 Exponents and logarithms 14](#_Toc298404136)

[2.2.2 Algorithms 15](#_Toc298404137)

[2.2.3 Exponentiation algorithms 15](#_Toc298404138)

[2.2.4 Modular exponentiation 16](#_Toc298404139)

[2.2.5 Tutorial exercises 17](#_Toc298404140)

[2.3 - Prime numbers and Euler’s totient function 17](#_Toc298404141)

[2.3.1 Prime numbers 17](#_Toc298404142)

[2.3.2 Euler’s totient function 17](#_Toc298404143)

[2.3.3 Factorisation 18](#_Toc298404144)

[2.3.4 Primality testing 18](#_Toc298404145)

[2.3.5 Tutorial exercises 19](#_Toc298404146)

[Section 3 – Secrecy, authentication and non-repudiation 20](#_Toc298404147)

[3.1 Secrecy 20](#_Toc298404148)

[3.2 Authentication 20](#_Toc298404149)

[3.3 Non-repudiation 20](#_Toc298404150)

[3.4 Tutorial exercise 21](#_Toc298404151)

[Section 4 – Simple and historical ciphers 22](#_Toc298404152)

[4. 1 The Caesar cipher 22](#_Toc298404153)

[4.1.1 Definition of the Caesar cipher 22](#_Toc298404154)

[4.1.2 Decrypting the Caesar cipher 22](#_Toc298404155)

[4.1.3 Tutorial exercises 23](#_Toc298404156)

[4.2 Substitution ciphers 23](#_Toc298404157)

[4.2.1 Substitution ciphers 23](#_Toc298404158)

[4.2.2 Tutorial exercise 24](#_Toc298404159)

[4.3 The Vigenère Cipher 24](#_Toc298404160)

[4.3.1 Vigenère cipher – the basic construction 24](#_Toc298404161)

[4.3.2 Le chiffre indéchiffrable 26](#_Toc298404162)

[4.3.3 Vigenère – an example 27](#_Toc298404163)

[4.3.4 Kasiski testing 28](#_Toc298404164)

[4.3.5 Tutorial exercises 28](#_Toc298404165)

[4.4 The Playfair cipher 28](#_Toc298404166)

[4.4.1 Implementing the Playfair cipher 28](#_Toc298404167)

[4.4.2 Encrypting a message 29](#_Toc298404168)

[4.4.3 Decrypting the Playfair cipher 30](#_Toc298404169)

[4.4.4 An applet 30](#_Toc298404170)

[4.4.5 Tutorial exercises 30](#_Toc298404171)

[4.5 Mechanical ciphers 30](#_Toc298404172)

[4.5.1 Rotor machines 30](#_Toc298404173)

[4.5.2 The Enigma machine 31](#_Toc298404174)

[4.5.3 Tutorial exercise 31](#_Toc298404175)

[Section 5 – Stream and block ciphers 32](#_Toc298404176)

[5.1 The one-time pad 32](#_Toc298404177)

[5.1.1 The basic idea 32](#_Toc298404178)

[5.1.2 Alice and Bob 32](#_Toc298404179)

[5.1.3 What actually is a one-time pad? 32](#_Toc298404180)

[5.1.4 Modular arithmetic, numbers and letters 33](#_Toc298404181)

[5.1.5 Encrypting using the one-time pad 34](#_Toc298404182)

[5.1.6 Decrypting using the one-time pad 35](#_Toc298404183)

[5.1.7 The perfect code 35](#_Toc298404184)

[5.1.8 Weaknesses of the one-time pad 35](#_Toc298404185)

[5.1.9 Historical use of the one-time pad 36](#_Toc298404186)

[5.1.10 Tutorial exercises 36](#_Toc298404187)

[5.2 Stream ciphers 37](#_Toc298404188)

[5.2.1 Definition and pseudorandomness 37](#_Toc298404189)

[5.2.2 Security and synchronisation 37](#_Toc298404190)

[5.2.3 Tutorial exercise 38](#_Toc298404191)

[5.3 Block ciphers 38](#_Toc298404192)

[5.3.1 Definition of block ciphers 38](#_Toc298404193)

[5.3.2 Arbitrary string lengths 38](#_Toc298404194)

[5.3.3 Binary numbers, bits and characters 40](#_Toc298404195)

[5.3.4 Tutorial exercises 43](#_Toc298404196)

[5.4 Data Encryption Standard and Advanced Encryption Standard 43](#_Toc298404197)

[5.4.1 Data Encryption Standard 43](#_Toc298404198)

[5.4.2 Advanced Encryption Standard 46](#_Toc298404199)

[5.4.3 Tutorial exercises 47](#_Toc298404200)

[Section 6 – Public Key Cryptography 48](#_Toc298404201)

[6.1 History and underlying ideas 48](#_Toc298404202)

[6.2 The RSA algorithm 49](#_Toc298404203)

[6.2.1 History and background 49](#_Toc298404204)

[6.2.2 An example 49](#_Toc298404205)

[6.2.3 The underlying mathematics 51](#_Toc298404206)

[6.2.4 Tutorial exercises 54](#_Toc298404207)

[Section 7 – Cryptography in electronic communications 55](#_Toc298404208)

[7.1 Digital signatures 55](#_Toc298404209)

[7.1.1 Integrity, authentication and non-repudiation 55](#_Toc298404210)

[7.1.2 Hashing 56](#_Toc298404211)

[7.1.3 Signing and timestamping 58](#_Toc298404212)

[7.1.4 Secure Hash Algorithm 60](#_Toc298404213)

[7.1.5 Tutorial Exercises 62](#_Toc298404214)

[7.1.6 Extended Tutorial Exercise 63](#_Toc298404215)

[7.2 Public Key Infrastructure 65](#_Toc298404216)

[7.2.1 Digital certificates 65](#_Toc298404217)

[7.2.2 Certificate Authorities 65](#_Toc298404218)

[7.2.3 PKI 66](#_Toc298404219)

[7.2.4 Tutorial exercise 67](#_Toc298404220)

[7.3 Electronic cash 67](#_Toc298404221)

[7.3.1 Cash 67](#_Toc298404222)

[7.3.2 Electronic cash 68](#_Toc298404223)

[7.3.3 Tutorial exercise 68](#_Toc298404224)

[7.4 Quantum cryptography 68](#_Toc298404225)

[7.4.1 History and background 68](#_Toc298404226)

[7.4.2 Photons and Uncertainty 69](#_Toc298404227)

[Section 8 - Cryptographic protocols and systems 70](#_Toc298404228)

[8.1 Protocols and systems 70](#_Toc298404229)

[8.1.1 Protocols 70](#_Toc298404230)

[8.1.2 Systems 70](#_Toc298404231)

[8.2 Some cryptographic protocols and systems 70](#_Toc298404232)

[8.2.1 Kerberos 70](#_Toc298404233)

[8.2.2 Pretty Good Privacy 71](#_Toc298404234)

[8.2.3 Secure Socket Layer 72](#_Toc298404235)

[8.2.4 Internet security 73](#_Toc298404236)

[8.2.5 Tutorial exercise 73](#_Toc298404237)

[Section 9 – Cryptanalysis 74](#_Toc298404238)

[9.1 Statistical, historical and brute-force attacks 74](#_Toc298404239)

[9.1.1 Statistical analysis 74](#_Toc298404240)

[9.1.2 Brute-force attacks 74](#_Toc298404241)

[9.1.3 Mathematical attacks 75](#_Toc298404242)

[9.1.4 Physical attacks 75](#_Toc298404243)

[9.2 Passive and active attacks 75](#_Toc298404244)

[9.2.1 Passive attacks 75](#_Toc298404245)

[9.2.2 Active attacks 76](#_Toc298404246)

[9.2.3 Tutorial exercise 76](#_Toc298404247)

[Section 10 – Conclusion 78](#_Toc298404248)

**Tutorial Exercises**

1.2.5 Introduction and overview................................................................................11

2.1.7 Modular arithmetic............................................................................................14

2.2.5 Exponentiation..................................................................................................17

2.3.5 Factorisation and primes...................................................................................19

3.4 Secrecy, authentication and non-repudiation.......................................................21

4.1.3 Caesar cipher.....................................................................................................23

4.2.2 Substitution ciphers...........................................................................................24

4.3.5 Vigenère cipher.................................................................................................28

4.4.5 Playfair cipher...................................................................................................30

4.5.3 Enigma..............................................................................................................31

5.1.10 One-time pad...................................................................................................36

5.2.3 Stream ciphers...................................................................................................38

5.3.4 Block ciphers and padding................................................................................43

5.4.3 DES and AES....................................................................................................47

6.2.4 RSA...................................................................................................................54

7.1.5 Digital signatures..............................................................................................61

7.1.6 Extended exercise.............................................................................................62

7.2.4 Digital certificates.............................................................................................66

7.3.3 Electronic cash..................................................................................................67

8.2.5 Protocols...........................................................................................................72

9.2.3 Attacks..............................................................................................................75

# Section 1 – Introduction and Overview

Before we get properly started, let’s look at what this module is all about, and what these notes are to be used for.

## 1.1 Purpose of these notes

It’s very important that you understand the purpose of these notes and what they contain.

### 1.1.1 Using these notes

These notes are supportive material only. You shouldn’t think that in any way they are a substitute for the lectures and tutorial classes. They are here to give you some indication of the content, but in lectures we will expand on the ideas, discuss aspects, spend time on those aspects which are proving difficult, and correct errors! In other words, these notes are a template from which the lecture classes will be delivered.

As we go through the module, we might introduce new things not contained here, we might omit some things contained here – we will take a fairly flexible approach.

Also, these notes are not a book and should not be treated as an authoritative guide – you will need to research for yourself to find out more about the subject.

### 1.1.2 Tutorial Exercises

After each concept introduced, there is a selection of tutorial exercises. Apart from the fact that you can use solutions to these exercises for your assignment submission, attempting them will greatly increase your overall understanding. There are a variety of exercises, some theoretical, some mathematical, some programming, and some that can be done in pairs or small groups. Written solutions will not be provided, but you can discuss your solutions with me and we will spend time on the concepts of the more difficult exercises.

## 1.2 Introduction to cryptography

What actually is cryptography? Why is it so important to keep information safe anyway? Why is it so difficult? And what does all this maths have to do with it? These are some of the questions we hope to answer as we go through this module.

In this module we’ll attack this subject by first of all looking at some of the basic concepts involved, and then looking at some well-known and some less well-known ways of coding information. In the second half, we’ll look more at current methods and techniques involved in trying to keep our data secure.

### 1.2.1 A Historical Context

Let us go way back in history, for example back to the days of the Roman Empire. Suppose we want to get a message from one base to another base giving details of an attack about to happen. How will we do it? There were no telephones, computers, fax machines, so probably we’d have to send a messenger from one base to the other, carrying details of the proposed attack.

What happens if an enemy intercepts him along the way? They can steal his piece of paper with the details and now the enemy knows exactly what the planned attack is going to be.

A better option would be for the messenger to carry the details of the attack written in such a form that only people of his own side can actually understand it. For example, suppose the two commanders of the two bases have developed their own secret language between them, which only they understand. The idea is that if they use that language in the message, then even if the messenger is intercepted, the attacker won’t understand the message and so not too much harm has been done.

But this still isn’t very safe. What if the enemy works out the secret language? What if someone gives away secrets about the language? How do they create the secret language in the first place – if it’s face-to-face, maybe someone is listening? If it’s by message, the messenger might get intercepted. It’s not exactly easy to keep secrets.

The challenge of coming up with “codes” – so a way of writing down what you want to say so that your friends can understand it but your enemies can’t – is the fundamental idea behind cryptography.

### 1.2.2 The Electronic Age

In the module we will look at some of the early, simple codes that were used. But the subject of cryptography really grew when electronic devices started to appear.

The telephone was invented in the 19th century. Now people could talk to each other across a phone line. So instead of writing down your secret plan, you could just call up your colleague and tell him it. Perfect. Or is it? As soon as a new invention comes along, someone will find a way to exploit it. Telephone works by sending signals across a cable. So an enemy could simply tap into the cable and listen to the entire conversation without anyone knowing. So we still need a code.

Nowadays most of us use computers. Many of us do online shopping, for example, which involves payment by credit card. To buy my product, I need to get my credit card details to the retailer. But I don’t want anyone malicious to intercept my message with my details, or they have all they need to go and use my credit card for their own purposes. We are all aware of the threat posed by computer “hackers”, viruses, etc, which is why we all install anti-virus software, because we aren’t 100% certain of transactions and communications via the internet.

Example: how many people would buy from the following (absolutely real) example of a web-site?

• You place your order

• You are told you can pay by sending a cheque or by credit card online

• You choose to pay online

• You are directed to a form asking you to fill in your credit card details

• These details are converted into a text file and you are asked to email it to the company as an attachment

Would you do this? Do you trust sending your credit card details as a text file by email? If someone intercepts it, they have all your details.

Probably 99.99% of people would not buy this way. They expect their details to be processed securely, in such a way that even if someone does hack into the process, they don’t get any useful information.

This is cryptography in the modern age. We expect our payments and transactions to be processed securely and safely. We need a way to get our credit card details from the customer to the seller in such a way that even if someone does hack into the system, they can’t find out the actual details. Basically, we need a code that is almost impossible to break.

This idea underpins the whole of this module.

### 1.2.3 Mathematics

All sounds fine, but what does mathematics have to do with it?

Quite apart from the philosophical argument that mathematics underpins everything in life, there are practical reasons for the mathematics we will study.

Applying our problems to computers, what actually is a computer? It’s a device that takes an input, performs some logical operations, and gives an output. The actual processing is done by binary switches. Everything in a computer is logical. That is mathematics.

The maths that we will study has particular importance. It’s quite easy to come up with maths problems that are really, really hard to solve, unless I give you a clue. For example:

*“I have thought of a number. Guess what it is.”*

Any ideas? That’s practically impossible, it could be anything. No-one is going to solve that. Suppose though you had the knowledge that no-one else had, that it’s an integer between 300,000 and 300,100. Who is likely to be the first person to guess it? You are! (Whilst everyone else is trying every number in the world, you try every integer between 300,000 and 300,100 which doesn’t take that long - there are only 101 to try!)

A lot of what we’ll study is mathematical problems that are easy to formulate, but very, very hard to solve unless you have a clue. So, you and your friend can work out the problem because you know the clue. But no enemy can. This forms the basis behind our codes. You’ll see the idea as we go through the module.

### 1.2.4 Summary

The main thing we are talking about in this module is the problem of getting information securely from one place to another, without the worry of dire consequences if someone intercepts our message. We will discuss ways of transmitting our messages in such a way that only the intended recipient can understand it, and if someone does try to intercept or “hack” our message, they don’t get anything out of it.

We will move on in the next few weeks to discuss the basic mathematical concepts you will need for the rest of the module.

### 1.2.5 Tutorial Exercise

For this week, just go onto the web and search for cryptography, have a look at a few pages and get a feel for some of the issues and things that are talked about. Don’t worry if it looks complex and bewildering, you’re just starting; we’ve a year of study to go! Just begin to get some sort of feel for the subject.

# Section 2 – Number Theory

Number theory will play a crucial role in the algorithms, ciphers, and protocols we will discuss in this module. We have to lay down these foundations at the start. The application to cryptography will not be immediately apparent at this stage, until we come on to ciphers in the next section, but these concepts are *essential.* You must make sure you understand them. Your first test will cover this section of the module.

Everything in this part of the module is concerned with the integers *Z* (in fact we will often restrict our attention to the natural numbers *N*).

## 2.1 Modular arithmetic

You are already familiar with one use of modular arithmetic even if you don’t know it!

### 2.1.1 The 24-hour clock

Consider the standard 24-hour clock, such as that used on train timetables. That means our times can range from 00:00 to 23:59. When we reach 24:00 we reset that to 00:00 and start counting again (imagine what times would be like if we didn’t reset, “I’ll meet you at 185634 o’clock”). For simplicity, we’ll forget about the minutes, and just consider the hours, so our hours range from 0 to 23.

Suppose it is 05:00, so the hours are 5. What will the time be in 8 hours? Pretty easy, it will be 13:00, so the hours are 13. We did this by working out 5 + 8 = 13 (it will get harder than this!) Now suppose it is 20:00, so the hours are 20. What will the time be in 6 hours? Well, it will be 02:00, so the hours are 2. When we added up 20 and 6, we remembered that when we reach 24, we go back to 0 again. So we have the odd-looking “equation” 20 + 6 = 2.

The main point is that we add up exactly as normal, but we “wrap around” back to 0 whenever we pass a certain point (in this case 24). So for example, suppose it is 07:00. What will the time be in 50 hours? In 24 hours it will be 07:00 again. A further 24 hours later it will be 07:00 again. Then another 2 hours later its 09:00 and our 50 hours have elapsed. In some sense, we have the “equation” 7 + 50 = 9.

### 2.1.2 Modulus

The equations in the previous section don’t really make sense; of course 7+50 isn’t 9! But it is when we “wrap around” at 24. The 24 here is called the **modulus** of the system we are talking about. We use the following notation:

7 + 50 ≡ 9 (mod 24)

This means that “wrapping around” at 24, 7 +50 is equivalent to 9. Note the use of the equivalence symbol ≡ rather than the usual sign = used for equality.

The example we have used here was 24, because it’s a familiar example. But what’s to stop us using any other modulus?

For example, take a modulus of 5. What is 3 + 4 (mod 5)?

With modulus 5, we wrap around when we reach 5, so the only numbers we have are 0, 1, 2, 3 and 4. When we reach 5 we go back to 0.

Trying to add 3 + 4, start from 3. Adding 1 gives 4. Adding another 1 gives 0 (we wrap back to 0 because we reached 5). Adding another 1 gives 1. Adding another 1 gives 2. So overall, adding 4 gives the answer 2, and 3 + 4 ≡ 2 (mod 5).

We can do multiplication as well: what is 3 \* 2 (mod 5)? (The \* symbol is commonly used for multiplication to avoid confusion with the letter *x*).

“Normally” 3 \* 2 is 6. But 6 is the “same as” 1 (try counting to 6 on the fingers of one hand) when we consider mod 5.

So we can say 3 \* 2 ≡ 1 (mod 5).

This can become clearer when we consider remainders.

### 2.1.3 Remainders

When you divide one number by another, you are probably used to the idea of *quotients* and *remainders*. For example, when you divide 27 by 4, the quotient is however many times 4 goes into 27 (in this case 6) and the remainder is what’s left over, in this case 3. This can be expressed as 27 = 6 \* 4 + 3.

How does this relate to modular arithmetic? Well, what is 27 (mod 4)? Start counting, remembering to wrap around every time we reach 4. We wrap around at 4, 8, 12, 16, 20, and 24 (so our 6 lots of 4), and then count up to 3 to reach 27. So 27 ≡ 3 (mod 4).

**Hence a number *x (mod n)* is equivalent to the remainder when you divide *x* by *n*.**

Another example: what is 57 mod 5? Well, the remainder when you divide 57 by 5 is 2 (57 = 11 \* 5 + 2). So 57 ≡ 2 (mod 5).

### 2.1.4 Congruences

Informally, two numbers *x, y* are **congruent (mod *n*)** - written *x* ≡ *y* (mod *n*) - if they have the same remainder when divided by *n*.

So, for example, 6 ≡ 11 (mod 5) since both 6 and 11 have remainder 1 when divided by 5.

Some other numbers that are congruent to 6 are 16, 21, 26, 31, 36, 41, …., so any number that has remainder 1 when divided by 5.

The “remainder” way of doing things becomes slightly trickier when dealing with negative numbers. What is the remainder when -4 is divided by 5? It’s not 4. Thinking about it, -4 = (-1)\*5 + 1 and so the remainder is 1. This makes sense following our pattern above. We’d already decided that 6, 11, 16, 21, 26, 31, 36, 41, …. are all congruent mod 5. Following the pattern downwards (difference of 5 between each number) we should have all of these congruent :

….., -39, -34, -29, -24, -19, -14, -9, -4, 1, 6, 11, 16, 21, 26, 31, 36, 41, ……

since they all have remainder 1 when divided by 5.

Just be careful with negative numbers!

Congruence can be given by a more formal definition. Suppose we have two numbers *a* and *b.* **Then we say that *a* ≡ *b* (mod *n*) if (*a – b*) is a multiple of *n*.**

This is equivalent to the above definition for the following reason:

Suppose *a* and *b* have the same remainder *r* when divided by *n*.

Then by definition of remainders, there are some *p* and *q* such that:

*a = p\*n + r*

*b = q\*n + r*

So (*a – b*) = *p\*n – q\*n = (p – q)\*n*

and so (*a – b*) is a multiple of *n.*

As an example, is 43 congruent to 59 mod 4? Well, 59 – 43 = 16 which is a multiple of 4 and so they are congruent mod 4 (in actual fact, they both have remainder 3).

### 2.1.5 A note on convention

We have used the symbol ≡ above to denote “equivalence”. So, for example, we said that

6 ≡ 11 (mod 5) because 6 and 11 are “equivalent” modulus 5 (they have the same remainder when divided by 5). There is one **and only one** case where it is convention to use a normal equals sign = which is when you are asked to give the “answer” to the question of what is a single number (mod *n*).

Example: What is 17 (mod 5)? There are many things that are congruent to 17 (mod 5), in fact any number with a remainder of 2 when divided by 5, so for example -13, -8, -3, 2, 7, 12, 22, 27, etc. The convention is to give the **smallest positive number**. So in this case we would say that 17 (mod 5) = 2. The “answer” to *x* (mod *n*) will always be a number in the range 0 to (*n* - 1), and will just be the remainder when dividing by *n*.

So, for example, 26 (mod 4) = 2 and 24 (mod 4) = 0.

### 2.1.6 Modular Relations

A crucial fact about modular arithmetic is the following theorem.

**Theorem 2.1.6**

If *a1 ≡ b1* (mod *n*) and *a2 ≡ b2* (mod *n*) then:

• *(a1 + a2) ≡ (b1 + b2)* (mod *n*)

• *(a1 \* a2) ≡ (b1 \* b2)* (mod *n*)

**Example**

2 ≡ 6 (mod 4) and 7 ≡ 15 (mod 4).

So (2 + 7) ≡ (6 + 15) (mod 4) and (2 \* 7) ≡ (6 \* 15) (mod 4), that is

9 ≡ 21 (mod 4) and 14 ≡ 90 (mod 4)

You can check this and see that it works, and you can try to prove the general result in the tutorial exercises.

### 2.1.7 Tutorial Exercises – Modular Arithmetic

Q1. Calculate the following:

(i) 19 (mod 5) (ii) -16 (mod 3) (iii) 28 (mod 7) (iv) 200001 (mod 10)

Q2. Calculate the following (a quick way to do it could use Theorem 2.1.6)

(i) 821 (mod 8) (ii) 320 (mod 8) (iii) -799 (mod 8)

Q3. What is the value of *x* (mod 1) for integers *x*?

Q4. (*More difficult):* Prove Theorem 2.1.6.

(Hint : If *a1 ≡ b1* (mod *n*) then they have the same remainder when divided by *n.* So, by definition, *a1 = p1 n + r1* and *b1 = q1 n + r1* for some *p1, q1.* Do the same for *a2 ≡ b2* (mod *n*). Now look at the remainders of *(a1 + a2), (b1 + b2), (a1 \* a2)* and *(b1 \* b2)* when divided by *n*.)

## 2.2 Exponentiation

One crucial aspect of cryptography arises when we consider exponentiation. Again, we will see the details of this with respect to cryptography later, and present the mathematical foundations here.

### 2.2.1 Exponents and logarithms

You should be familiar already with exponents and logarithms.

Exponents are problems such as working out 25 = *x*. In this example, it’s fairly straightforward to do 25 = 2 \*2 \* 2\* 2 \* 2 = 32.

Logarithms are the related problems such as working out 2*x* = 32. In this case, the solution *x =* 5 is pretty clear from the above, but it’s not that easy to work out if you didn’t already know it, (you didn’t have the clue from above).

In general, we use the notation *logxy = z* to mean that *xz = y*. So for example,

• log39 is 2 because 32 = 9.

• log101000 is 3 because 103 = 1000.

• log93 is 1/2 because 91/2 = 3 (recall that a power of 1/2 represents square root).

The basic fact is that, in general, it is easier to work out exponents than it is to work out logarithms. Remember our fundamental idea of cryptography – we want something we can work out easily, but enemies can’t easily work out. These two problems above are closely related (and in some sense are opposite to each other). All we have to do is work out exponents. Our enemy, to try and hack us, has to decode the problem (that is to work out logarithms), which is harder. That gives us an inkling that this idea might be useful for cryptography, but again the details of this are for later in the module.

### 2.2.2 Algorithms

Here’s an issue we haven’t really touched on yet. Given a problem, there are different ways to go about solving it. What makes one way better than another? As long as we eventually get the right answer, does it matter which way we choose to go about it?

Take a practical example – finding a word in the dictionary. Let’s suppose we want to know what “syzygy” means (it’s a real word, I promise you, looks great in Scrabble!) Here is one way to go about finding it:

• Start at the first word in the dictionary (“a”)

• If that’s not our word, move onto the second word (probably “aardvark”)

• If that’s not our word, move onto the third word (maybe “abacus”)

• And so on…..

And here is another way:

• Guess roughly where “S” is in the book

• Check where you turn to and decide if you need to go forward or back to find “syzygy”

• Do the same again

Which approach would you take? Would you take Approach 1 - go through all the A’s then all the B’s in turn until you got there? Or would you take Approach 2 – open the dictionary at about S, move forward a bit, maybe back a bit, until you found it?

The reason we take the second approach is simple – it is quicker. It is a better way to search for the word.

Analysis of algorithms is important – we need to make sure that there are **no** easy ways for hackers to get into our system and decode our messages, and we need to make sure we are doing our work as efficiently as possible and not wasting time that an enemy could be exploiting.

### 2.2.3 Exponentiation algorithms

Suppose I want to work out (for whatever reason) 264.

Here’s one way to do it:

• Well, that means 2 \* 2 \* 2 \* 2 \* ……. \* 2 \*2 (64 times)

• So start with 2

• Then multiply it by 2, I’ve done 2 \* 2 = 4

• Then multiply that by 2, I’ve done (2 \* 2) \* 2 = 4 \*2 = 8

• And so on……..

This will take 64 steps (I haven’t written then all out!).

Here’s another way:

• 264 = (232)2 by usual power laws

• And then 232 = (216)2 so 264 = ((216)2)2

• And then 216 = (28)2 so 264 = (((28)2)2)2

• And then 28 = (24)2 so 264 = ((((24)2)2)2)2

• And then 24 = (22)2 so 264 = (((((22)2)2)2)2)2

• And then 22 = (2)2 so 264 = ((((((2)2)2)2)2)2)2

• Which we work out

This only takes 6 basic steps to work out, we square 2, and then square that answer, then square again, 6 times.

Which is quicker, which is better? (if you remain to be convinced, write out Algorithm 1 in its full 64 steps).

OK, for modern computers, the difference between 6 steps and 64 steps isn’t that great. But in current systems, the numbers we are talking about won’t be of the order 2 and 64, they will be hundreds of digits long, numbers that are actually very hard for the human mind to comprehend their magnitude. Then it starts to make a difference. On current computing power, we are talking about the difference between a few seconds and several thousand years to solve these. Still think it doesn’t make a difference which algorithm we choose? :)

### 2.2.4 Modular exponentiation

In calculating such numbers, one has to be careful of the amount of memory used as well as the number of steps. I’m sure you’ve all had the frustration of your computer running painfully slow while its memory tries to do too many things at once.

As a simple example, try to work out 320 (mod 137). These are *tiny* numbers compared to the numbers used in modern computer systems.

I can work out 320 eventually using one of my algorithms from above. The answer is 3486784401. Then working that out (mod 137) on a computer gives 4. The trouble is, we’ve used a huge amount of memory storing such a big number to get to this answer.

A better way would be to note what we stated in Theorem 2.1.6. It allows us to reduce the numbers to their modular equivalent at every step. So for example:

320 (mod 137) ≡ (35)4 (mod 137) = (243)4 (mod 137) = (106)4 (mod 137)

because 243 ≡ 106 (mod 137)

Keeping going, we can solve the problem without ever needing to store a number more than 3 digits long.

We’ll investigate this further later on – for now, all you need to understand is that by applying tricks and sensible algorithms, we can save both memory and time and solve our problems quickly and efficiently.

### 2.2.5 Tutorial exercises

Q1. Work out the following:

(i) 24 (ii) 53 (iii) 103

Q2. Work out the following:

(i) log525 (ii) log216 (iii) log2(1/2)

Q3. Go home, pick up your phone book, and look up a friend or a company’s number. (This is very similar to looking for a word in the dictionary as in the example above). Which method did you choose to find them? Why did you find them this way and not some other way?

Q4. Work out 564 (mod 24). *There is an easy way to solve this!*

Q5. What is 5327 (mod 2)? *There is an easy way to solve this!*

## 2.3 - Prime numbers and Euler’s totient function

In this section we will look at prime numbers. Again the application to cryptography will become clearer later on.

### 2.3.1 Prime numbers

An integer *p* is said to be **prime** if the only integers that divide exactly into *p* are 1 and *p.* For technical reasons, 1 is not considered to be prime, so the first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,…...

Any number that divides exactly into a number *n* is called a **factor** of *n.* So for example the factors of 16 are 1, 2, 4, 8, and 16.

Two numbers are said to be **coprime** if the only factor they have in common is 1. So for example 5 and 21 are coprime, but 6 and 21 are not, since they share a common factor 3.

### 2.3.2 Euler’s totient function

Euler’s totient function (named after the famous 18th century Swiss mathematician Leonhard Euler, widely regarded as one of the greatest mathematicians of all time) can be defined as follows:

Given a positive integer *n*, the **Euler totient function** φ (*n*) is the number of positive integers less than *n* which are coprime to *n*. It is sometimes called the **phi function** (the symbol used is a lower-case Greek letter phi). It is extremely important in various branches of mathematics.

As an example, what is the Euler totient function of 8? The numbers less than 8 that are coprime to 8 are 1, 3, 5 and 7, so there are four of them. The other numbers (2, 4, and 6) are not coprime to 8 as they share a common factor of 2. So φ (8) = 4. Similarly the integers less than 12 that are coprime to 12 are 1, 5, 7, 11 and so φ (12) = 4 as well.

Note that for a prime number *p*, the only number that is not coprime to *p* is 1. (If there was a number that was not coprime to *p* then it would, by definition, share a factor of *p* which is impossible since *p* is prime). Hence φ(*p*) = *p* – 1 for any prime *p*.

### 2.3.3 Factorisation

It is a fundamental mathematical fact that every number can be “factorised” into primes in essentially only one way. What does this mean? It is best illustrated by examples:

28 = 2 \* 2 \* 7 - all the factors are prime numbers.

42 = 2 \* 3 \* 7 - all the factors are prime numbers.

35 = 5 \* 7 - all the factors are prime numbers.

By “essentially” we mean that there is no other way to factorise the number, apart from just writing the factors in a different order (e.g. 35 = 5 \* 7 and 35 = 7 \* 5 but this is really just the same factorisation, involving the same numbers).

Writing a number *n* in this way, so in the form n = *p1* \* *p2* \* *p3* \* ….. where the *p1* are primes, is called the *prime decomposition* of *n*.

The problem of factorising a given number is computationally a “difficult” problem and will be used extensively in some of the cryptographic methods we will discuss as the module progresses.

### 2.3.4 Primality testing

Prime numbers are a fundamental building block of mathematics and have many cryptographic applications. How can we check to see if a given number *n* is prime?

The naive, easy algorithm, is as follows:

• Start with 2. Does 2 divide *n*? If so, it cannot be prime

• Then check 3. Does 3 divide *n*? If so, it cannot be prime

• Then check 4, then 5, then 6, and so on until we reach *n*.

This certainly works, but it takes a fairly long time – we have to check *n* numbers. A few observations can make this a lot simpler.

• We don’t have to go all the way up to *n*. In fact, it is enough to go up to *n* (basically if a factor is more than *n* then it must multiply by a number smaller than *n* to make *n* and so we would have already found it).

• We only need to check prime numbers (for example, if we checked 2, then we don’t need to check 4, 6, 8 etc as they cannot possibly divide *n* if 2 doesn’t).

• All prime numbers (except for 2 and 3) are of the form 6*m* + 1 or 6*m* – 1 for an integer *m*. You can prove this in the tutorial exercises.

So, what we can do is store in a database a list of the first few primes and then just use those to test our number and see if any of them divide into *n*.

There are many algorithms that can be used. There are some “probabilistic” algorithms that work very fast, and though cannot absolutely prove a number to be prime, can tell us that the number is “almost certainly” prime. These are really outside the scope of this module but feel free to do some research if you wish to!

### 2.3.5 Tutorial exercises

Q1. Factorise the following numbers into primes:

(i) 36 (ii) 125 (iii) 1000 (iv) 16

Q2. Give the Euler totient function φ(*n*) for the following numbers:

1. 14 (ii) 9 (iii) 24 (iv) 37

Q3. Show that any prime number must be of the form 6*m* + 1 or 6*m* – 1. (Hint: what would be a factor if it was of the form 6*m* + 2 or 6*m* + 3 and so on?)

Q4. (*Programming knowledge needed*) Write a short program, in a language of your choice, which allows the user to input a number and then tests to see if that number is prime. You can use the naive algorithm, but think about any improvements you might be able to make.

# Section 3 – Secrecy, authentication and non-repudiation

In this brief section we discuss the fundamentals behind computer communication and why cryptography is needed at all.

## 3.1 Secrecy

Why do we need secrecy? Shouldn’t we all just be open and honest about everything? The problem is that there are always people who will use information for the wrong purposes. The government of a country, of course, does not want the details of its weapons to be known to any enemies. I doubt very much if any of you would be happy to walk around with a big sign with all your credit card details on for anyone to see. Or detailing your medical history? The fact is, that there are some things that we prefer to keep to ourselves. So when we conduct communications we want to make sure that these private things stay private, and aren’t seen by everybody.

That is one of the fundamentals of cryptography – even if someone does get access to these details, it is in such a form that they can’t read it anyway, and so it stays a secret.

## 3.2 Authentication

Authentication is the technical word used for the practice of confirming someone’s identity. Again, you don’t want some random stranger to pretend to be you, and access your bank account. You expect the bank to have some security procedures in place to ensure that transactions are carried out by you. For example, cheques take four days to clear because they check the signature matches the one on your account. When you call the bank, you are asked security questions, such as your date of birth, mother’s maiden name, and so on. When you use your cash card you use a PIN number that only you know. When you log into a computer account, you have to supply a password. The idea is that you are the only person who knows the password, or whatever security details are requested, and so someone cannot pretend to be you.

Of course, no system is absolutely 100% foolproof. Someone might get hold of your password, or just guess it (that’s why it’s important to choose your password sensibly so as to be almost unguessable). But the aim of authentication is to be, as sure as you can, that the person making the communication is the person you believe it to be.

## 3.3 Non-repudiation

Repudiation means to refuse to acknowledge a fact. For example, you might run up a huge credit card bill and then refuse to accept that you made the transactions and refuse to pay the bill. Of course, the credit card company wants to avoid this, so they want to prove that you did make the transactions. From a customer point of view, if an order is placed and the goods never arrive, the customer wants the assurance of knowing that their order was placed and they are legally entitled to their goods.

A lot of commerce nowadays is carried on via the internet. A customer doesn’t sign for anything or have any face-to-face meetings with the vendor. The aim for both the customer and the vendor is to make sure that the “contract” which supplies the goods from the vendor to the customer, is made between the correct people.

A fraudster might get your credit card details and order things in your name. Non-repudiation is all about making sure that the people who make the contract are indeed the people who want it to be made.

So non-repudiation works in two ways – the customer has an assurance that they have placed the order and are legally entitled to their goods, and the vendor has confirmation that the order was placed and that they are entitled to your payment.

## 3.4 Tutorial exercise

Do some research into these concepts and find some examples of where each of them have been breached. There is no need to write more than a few lines for each as long as you convey your understanding with a suitable example.

# Section 4 – Simple and historical ciphers

We have now completed the basic background needed to take our study of cryptography further. We will now move on to investigate some well-known, and some less well-known, codes and ciphers and start to understand the basis behind encrypting and decrypting data.

## 4. 1 The Caesar cipher

The Caesar cipher is one of the first examples of coding being used in communications, and was used by Julius Caesar to communicate with his colleagues. It is very simple, but was reasonably effective at the time (it is worth bearing in mind that most people at that time were illiterate and had never seen cryptography before so were unlikely to crack the code even though it is very simple).

### 4.1.1 Definition of the Caesar cipher

The idea behind the cipher is simple. Take your original message, and then move each letter through the alphabet by a fixed amount. So for example, let’s consider the Caesar cipher with a shift of 1. This means A goes to B, B goes to C, and so on.

Hence the phrase “ATTACK AT DAWN” would become “BUUBDL BU EBXO”

If we used a shift of 3, it would become “DWWDFN DW GDZQ”

If we used a larger shift, we have to note that if we go past Z, we wrap back round to A.

So with a shift of 5 the coded message is “FYYFHP FY IFBS”

This is analogous to modular arithmetic when we “wrapped round”. If we label our alphabet with the numbers 0 to 25 (A is 0, B is 1, and so on) then to make our coded message, for each letter *l* we simply work out *l + s* (mod 26) where *s* is our shift.

The procedure of converting our message to a coded form is called **encryption**.

### 4.1.2 Decrypting the Caesar cipher

It’s not really that hard to solve the Caesar cipher. The process of taking a coded message and recovering the original message is called **decryption.**

The easy way to decrypt our message is just to try every possible shift. Remember that when decrypting, we want to do the opposite of what made the code in the first place.

So, for example, take “DWWDFN DW GDZQ”.

Let’s try with a shift of 1. This would mean the original message would have been “CVVCEM CV FCYP”. That doesn’t make any sense. So it’s not a shift of 1.

Now try a shift of 2. That would mean the original message was “BUUBDL BU EBXO” which again makes no sense. So it’s not a shift of 2.

So try a shift of 3. That would mean the original message was “ATTACK AT DAWN” which makes sense. We cracked the code and worked out the original message!

The most we’d have to do to crack this code is to try it 25 times for each possible shift (a shift of 26 is the same as doing nothing). Doing something 25 times is no effort at all, especially with computers at our disposal.

In practice we wouldn’t even try all 25 shifts. We can use probability to guess what the shift might be. For example, the most common English letter is E. So in a long message we might count which letter appears the most times, guess that it represents E, and work out the shift accordingly. It might not work, but there’s a good chance it will. We’ll revisit this idea several times in the module.

### 4.1.3 Tutorial exercises

Q1. Encrypt the following messages with a shift of 3:

(i) “CAESAR CIPHERS ARE EASY”

(ii) “THIS IS A SECRET MESSAGE”

Q2. Decrypt the following messages (try various shifts until you find a sensible phrase):

(i) “ETARVQITCRJA KU HWP”

(ii) “RSXXMRKLEQ JSVIWX JSSXFEPP GPYF”

Q3. Give two possible decryptions (real English words) of the code “BMJJQ”

Q4. (*Advanced – programming and string manipulation skills needed)* Write a program that allows you to type in a phrase and a shift then encrypts the phrase by that shift, and then allows you to type in a coded phrase and prints out all the possible decryptions of that phrase.

## 4.2 Substitution ciphers

The Caesar cipher is an example of what is known as a *substitution cipher*, that is a cipher where every letter is replaced by another.

### 4.2.1 Substitution ciphers

In the Caesar cipher, with shift 3 say, every letter is replaced by the letter 3 further on in the alphabet. We have the following set of substitutions, where in our encryption, the letter on the top is replaced by the letter on the bottom.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |

Another type of substitution cipher wouldn’t just have a cycle of replacement letters, but some randomly chosen permutation. For example, we might have

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| D | I | H | F | V | O | P | A | J | Z | Q | E | W | M | G | N | X | C | L | Y | T | U | B | S | R | K |

Under this substitution, “ATTACK AT DAWN” becomes “DYYDHQ DY FDBM”

There are potentially 26! (that is, 26 \* 25 \* 24 \* …… \* 3 \* 2 \* 1) possible substitution ciphers (26 choices for A, then 25 choices for B, and so on) which is a lot. There are two major problems with this approach:

• Both parties in the transmission need to know the exact permutation. This is hard to keep secret: if an attacker manages to get hold of it, then their decryption is easy.

• A particular letter is always encrypted to the same letter each time. For example above, A is always encrypted to D. This makes the challenge of decrypting somewhat easier. For example, as we said before, E is the commonest letter in the alphabet, so if we have a lot of one particular letter appearing in our text, there’s a good chance its E. If there is a one-letter word in the text, then that must correspond to A or I, as they are the only one-letter words in English. The most common word in the English language is “THE”. So if we have the same three-letter encoding many times, it’s quite likely to be “THE”. We will see this again.

### 4.2.2 Tutorial exercise

Have a go at the following example. Use the ideas from above, and your knowledge of the English language, to try and work out some of the letters. Once you have a few, the rest should start to slot into place. It is a simple substitution cipher - only letters are encrypted - the punctuation symbols are not encrypted, so the ? is really a ?, for example.

HX KJH HTTEUZHXU Z AENL FLLX EUDLB, “IQES MQ FEFFEYL, ZV SHO IOK ZXKH KAL METAZXL JQHXY VZYOQLU, JZGG KAL QZYAK EXUJLQU THML HOK?” Z EM XHK EFGL QZYAKGS KH EIIQLALXB KAL DZXB HV THXVOUZHX HV ZBLEU KAEK THOGB IQHNHDL UOTA E COLUKZHX.

Feel free to work on this with friends. This quote is from Charles Babbage, renowned for being the inventor of the first computer.

## 4.3 The Vigenère Cipher

The Vigenère (pronounced something like Veezh-nare) cipher wasn’t actually invented by Vigenère at all. It was originally put forward by Giovan Batista Belaso in 1553. Vigenère also published it, but some time later in the 16th century, however his name as inventor was misattributed to the cipher in the 19th century and it has stuck. It has a crucial advantage over substitution ciphers, in that a letter isn’t always encrypted to the same letter each time, so the tactics used above are no longer usable.

### 4.3.1 Vigenère cipher – the basic construction

The Vigenère cipher utilises the following table, which consists of A to Z written along the top and left hand side, and then the letters from A to Z shifted on each line (shifted by 0 in the first row, by 1 in the second row, and so on, up to a shift of 25 in the last row).



The procedure to follow is as below:

* Write down the phrase you wish to encrypt, for example “ATTACK AT DAWN”
* Both parties have already chosen a secret keyword, let’s say for example they chose “BREAK”
* Write down the keyword over and over again underneath the phrase, like this :

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | T | T | A | C | K | A | T | D | A | W | N |
| B | R | E | A | K | B | R | E | A | K | B | R |

* Start with the first column. We have B underneath A. So look up in the big Vigenère table above, the letter that corresponds to row B and column A. It’s B. So write that down. Then the second column, we have R underneath T. So look up the letter that corresponds to row R and column T. That letter is K. Next column, we have E underneath T. So look up the letter in row E and column T. It’s X. Continue to the end. The encrypted message is “BKXAML RX DKXE”

Now decrypting is easy for our friend because they know the keyword. The procedure is as follows:

• Write down the encrypted phrase and then the keyword underneath it like before.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B | K | X | A | M | L | R | X | D | K | X | E |
| B | R | E | A | K | B | R | E | A | K | B | R |

* For each pair of letters, look at the big table and:
  + Go to the row labelled by the keyword letter.
  + Find the encrypted letter in that row
  + Write down the column in which it appears
* So for example :
  + The first pair is B-B. Go to row B (the keyword letter). B (the encrypted letter) is in the column labelled by A. So we write down A.
  + The second pair is K-R. Go to row R (the keyword letter). K (the encrypted letter) is in the column labelled by T. So we write down T.
  + The third pair is X-E. Go to row E (the keyword letter). X (the encrypted letter) is in the column labelled by T. So we write down T
  + And continue….we do get “ATTACK AT DAWN”

Note the crucial difference here if we are trying to attack this message but don’t know the keyword. The first A is encrypted to B, but the second one is encrypted to A, and the third to R and the fourth to K. This means that we cannot do an analysis like before, and look for common letters appearing.

This means that the code appears almost impossible to crack unless you know the keyword. So is it really as perfect as it seems?

### 4.3.2 Le chiffre indéchiffrable

For a long time, the Vigenère Cipher was considered unbreakable. It was known as “le chiffre indéchiffrable” which is French for “the unbreakable cipher”. It was believed that is was essentially impossible to crack.

However, it has one fundamental weakness which eventually led to it being broken, which is the repetitive nature of the keyword. Have a look at the example above, with phrase “ATTACK AT DAWN” and secret keyword “BREAK” which has length 5. The thing to note is that the encrypted letters in positions 1, 6, 11 (corresponding to the B in the keyword) are obtained by a simple Caesar cipher of shift 1. The encrypted letters in positions 2, 7, 12 (corresponding to the R in the keyword) are obtained by a simple Caesar cipher of shift 17. So what we have is just a sequence of Caesar ciphers. If the attacker can guess the length of the keyword, then it’s just a matter of solving a few Caesar ciphers. It’s very easy for a computer to just guess at the keyword length and then try to solve the Caesar ciphers, doing this systematically would not take a computer much time.

This is probably easier to see by example, so I’ll work through an example of decrypting a message and then you can have a go.

### 4.3.3 Vigenère – an example

Here’s a message that we’ve intercepted. What could it mean?

RTTHWV TMBLQSGZ IOEK K VTLH R JWILP IQWB MSI VPT TIVGBCN

Let’s suppose we managed to guess (maybe we tried other lengths and they didn’t seem to work) that the keyword is 6 letters long. **We don’t know what the keyword is, we’re just guessing it’s of length 6.** Can we crack this code?

Our first tactic is to look at the words of length 1. There are two of them, and they can only be an I or an A. So there are four possibilities here (I-I, I-A, A-I and A-A). We could try each of these. Let’s just suppose we hit lucky first time (if it didn’t work we’d just try another possibility, would take a bit longer but still easy for a computer). We think that the first one-letter word is I and the second one is A (sounds sensible, something like “I want a”, “I like a”, maybe?).

So, we think that the “K” on its own represents an “I”. This is a shift of 2. Since we guessed the keyword length to be 6, we can apply a shift of 2 ever time we repeat the keyword, that is every six characters.

It leaves us with:

**P**TTHWV **R**MBLQS**E**Z IOEK **I** VTLH R **H**WILP I**O**WB MSI **T**PT TIV **V**QCN

And we guessed that the “R” on its own represents “A”. That would correspond to a shift of 17. So apply this every six letters as well

**P**TTHW**E R**MBLQ**BE**Z IOE**T I** VTLH R**A H**WILP **RO**WB MS**R T**PT

TI**ET**QCN

Right. What now? We’re going to use our knowledge of the English language. Look at the penultimate word. It’s three letters long and begins with “T”. What’s the most common word with three letters beginning with “T”? There’s a very good chance that word is “THE”. Let’s try it. That would mean the “P” represents “H” which is a shift of 8. We get

**PL**THW**E RE**BLQ**BER** IOE**T I N**TLH **A HO**ILP **ROO**B MS**R TH**T

TI**ETI**CN

And completing our “THE” we need a shift of 15. This gives us

**PLE**HW**E REM**LQ**BER T**OE**T I NE**LH **A HOT**LP **ROOM** MS**R THE** TI**ETIN**N

This looks good. We’ve got a word “ROOM” spelt out which is promising. What now? Well, the last word ends in “IN\*” (where we don’t know what the star is). Looks a very good candidate for an “-ing” word - we know how common they are. That would be a shift of 7. So do that every 6 places.

**PLEA**W**E REME**Q**BER TH**E**T I NEE**H **A HOTE**P **ROOM F**S**R THE M**I**ETING**

It’s easy now, the last word is obviously “MEETING” so that’s a shift of 4 for all the remaining characters

**PLEASE REMEMBER THAT I NEED A HOTEL ROOM FOR THE MEETING**

The important thing is that we didn’t even look what our keyword was, we worked out the answer anyway. As a matter of fact, counting the various shifts, which cycle in the order 2 – 8 – 15 – 7 – 4 – 17 we see that these correspond to the letters C – I – P – H – E –R and so our keyword was “CIPHER”. But we didn’t need it to crack the code, we just needed to guess how long it was!

### 4.3.4 Kasiski testing

In the example above we “guessed” a keyword length of 6. Are there any ways to help our guesses? One well-known way is the Kasiski method.

This works on the fact that certain words appear very often in English (words such as “the” or “and”). In a long message, where the keyword is repeated many times, it’s quite likely that they will end up being encrypted in exactly the same way somewhere along the line.

The Kasiski method is to search through and look for repeated patterns. Let’s suppose for example, that we have a repeated pattern 30 letters apart. Then there is a good chance that these are the same word encrypted in the same way, so using the same letter of the keyword. Hence the key length is likely to be a factor of 30, so it might be 2, 3, 5, 6, 10, 15, 30? (the idea is that if the key is 10 letters long, say, 30 letters after the first instance, we have cycled through the keyword three times and are back at the same place).

Now suppose there is another repeated pattern 35 letters apart. Then the keyword has a good chance of having length a factor of 35, so 5, 7, 35?

Taking these two observations together, it looks a good chance that 5 could be the keyword length since it works for both repeated patterns. Of course, it might not be, these patterns might be just coincidence, but it’s a sensible, informed guess.

### 4.3.5 Tutorial exercises

Q1. Using the keyword “VICTORY”, encrypt the phrase “WE ARE THE CHAMPIONS OF THE WORLD” using the Vigenère cipher.

Q2. Decrypt the following message (Hint: use Kasiski testing to spot similar patterns (words that look identical) and from the distance between them, guess the length of the keyword. Then use similar techniques to the above example).

L IE QB WPW VCS WX VVH PANZ. L AMIUHAL YS DBLCQN BG VVH EWUH.

## 4.4 The Playfair cipher

The Playfair cipher was invented in 1854 by Charles Wheatstone, but takes its name from Lord Playfair, who was heavily involved in promoting its use. It moves us on one step from our previous ideas, in that it deals with encrypting *pairs* of letters, not just single letters. This is a big difference to our previous ciphers. Then, there were 26 possible encryptions of a letter. Now, there are 26\*26 = 676 possible encryptions of a pair of letters (actually this isn’t quite true as you will see, but the idea is there) so it is substantially more difficult to crack.

### 4.4.1 Implementing the Playfair cipher

There are a few variations on this but I’ll describe a common implementation. It relies on a keyword and then a 5 x 5 grid containing letters. Immediately you spot a slight problem – there are 26 letters but only 25 spaces. It was common to incorporate I and J together in one square as there are virtually no words where it is not obvious whether we want I or J. Omitting Q was also occasionally used.

Having decided on your secret keyword, write down the keyword in the first few spaces in your 5 x 5 table (start in the top left and work your way across, then onto the next row and so on), and then fill in the rest of the letters alphabetically. Each letter can only appear once, so if we’ve already written it down we ignore a further occurrence.

This is clearer with an example. Let’s suppose our secret keyword is “HEREFORD”. We write down the letters in our keyword, so H, E, R, but next is E and we’ve already got that, next is F then O so we write them down. Then we have R and we’ve already got that, so we write down D. Then fill in the rest of the alphabet. We get this table (remember that I and J are combined):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| H | E | R | F | O |
| D | A | B | C | G |
| I | K | L | M | N |
| P | Q | S | T | U |
| V | W | X | Y | Z |

### 4.4.2 Encrypting a message

Now take our message, let’s say it’s “MEET ME AT NOTTINGHAM STATION”. We want to split this phrase up into pairs of letters, but we must be careful we don’t get any “double pairs”, that is a pair like “AA” or “BB”. So first of all, go through the message and between every identical pair of letters, just put an “X”. So we get

“MEXETMEATNOTXTINGHAMSTATION”

which when split in pairs gives

“ME XE TM EA TN OT XT IN GH AM ST AT IO NX”

Now go through each of the pairs and find each letter in the table above. There are three possibilities:

• If both letters are in the same row, replace them by the letters immediately to their right (if one of them is on the end of the row, wrap back around to the beginning of the row)

• If both letters are in the same column, replace them by the letters immediately below them (if one of them is on the bottom of the column, wrap back around to the top of the column)

• Otherwise, imagine drawing a rectangle which has the two letters as opposite corners, and write down the two letters that form the other corners of the rectangle (starting with the corner on the row corresponding to the first of the two letters).

Let’s go through this in detail with our message “ME XE TM EA TN OT XT IN GH AM ST AT IO NX”

ME – on different rows so imagine the rectangle with M in one corner and E in the other. The other corners of this rectangle would be K and F, so write KF.

XE similarly gives WR

TM – these are on the same column so look at the letters immediately below T and M. We get YT

EA – similarly gives AK

TN - rectangle, get UM

OT – rectangle, FU

XT – rectangle, YS

IN – these are on the same row so look at the letters immediately to the right of I and N. We get KI

GH – rectangle, DO

AM – rectangle, CK

TS – same row, take letters to the right, TU

AT – rectangle CQ

IO – same column, take letters below, NH

NX – rectangle, LZ

So our encrypted message is “KF WR YT AK UM FU YS KI DO CK TU CQ NH LZ”

### 4.4.3 Decrypting the Playfair cipher

With modern computing power, the Playfair isn’t too difficult to decrypt. Even with over 600 possibilities for pairs, this isn’t too challenging for a computer.

One technique is to use a similar strategy to before, but instead of looking for “E” as the most common letter, look for the most common pairs of letters in English (e.g. “ED” is very common at the end of a word).

It is possible now to use such “brute force” algorithms (essentially checking every possibility) because of the speed of modern computers, but back in the 19th century, this was an inconceivable task.

Once you have found the keyword, decryption is simple.

### 4.4.4 An applet

The following webpage has a nice applet to encrypt and decrypt using the Playfair cipher: **http://www.simonsingh.net/The\_Black\_Chamber/playfaircipher.htm**

### 4.4.5 Tutorial exercises

Q1. Without using any tools from the web (type this up in a word processor), encrypt “MATHEMATICS POSSESSES NOT ONLY TRUTH BUT SUPREME BEAUTY” (this is a quote from Bertrand Russell) using the Playfair cipher with keyword “PLAYFAIR”. Give all of your working

Q2. Check your answer on the applet given above, and try a few other phrases and keywords to be sure you understand how this cipher works.

## 4.5 Mechanical ciphers

The ciphers that we have looked at so far are all “pen-and-paper” ciphers, you can sit with a pen and paper and try to decrypt them. But as technology advanced, especially in the early 20th century, machines started to be able to be used. Remember then there were no computers, no internet or anything like that, but there were mechanical and electrical devices that started to become useful for cryptographic purposes. The most famous of these is undoubtedly the Enigma machine.

### 4.5.1 Rotor machines

This is not an electronics module and so we will give only the basic ideas behind the construction of mechanical machines for cryptography. Rotor machines were particularly in vogue in the middle of the 20th century, before computers were really invented.

Remember that one of the important factors about the ciphers we have talked about so far, is that those that are more difficult to crack are those that don’t necessarily encrypt a letter to the same letter each time. We’ve done this by using pen-and-paper techniques with keywords. But with electrical wiring we can automate much of this process and so make it even more difficult to crack.

A *rotor* is a rotating wheel with a combination of letters, with electrical contacts attached. As the rotor revolves, it changes the wiring amongst the electrical contacts and so the encryption is changed. This automates the process of changing the substitution at each step.

By using more than one rotor, connected via a gearing system, the automation can become very complex. For example with three rotors connected together, where the second rotor advances by one place after the first rotor has completed a full cycle of 26 turns, and similarly for the third rotor, your “key length” before you go back to the start can be up to 26 \* 26 \* 26 = 17, 756 – compare that with our key length of 5 or 6 earlier – and everything is automatic!

Many such rotor systems were designed to be symmetric – repeating a set of instructions got us back to the start so decryption was easy for those who knew the key.

### 4.5.2 The Enigma machine

The Enigma machine was a particular type of rotor machine. There were many variants of this machine, the standard well-known example used three rotors. The second rotor would step an additional place every time the third rotor moved on, so there would be a form of “double-stepping” going on, which made the key additionally hard to discover.

The most well-known version of the machine incorporated a *reflector* – this ensured that decryption followed the same procedure as encryption. However it did entail including a serious cryptographic flaw; that it was impossible for a letter to encrypt to itself.

The most famous use of the Enigma machine was by the Nazi German Army in the Second World War. The Allied Forces’ eventual cracking of the machine (with much work being conducted at Bletchley Park, near Milton Keynes, UK) proved to be fundamental to the war effort, and is estimated to have accelerated the ending of the war by up to two years, although of course such figures can only ever be hypothetical.

A 2001 film *Enigma*, starring Kate Winslet, tells the story of the Enigma machine but has been derided for a series of factual and historical inaccuracies.

### 4.5.3 Tutorial exercise

Go onto the web and find out what you can about the Enigma machine. Write around half a page to a page highlighting its major features and any interesting facts you can discover.

# Section 5 – Stream and block ciphers

In this section, we will look at two related concepts of ciphers, that is the idea of stream ciphers and block ciphers.

## 5.1 The one-time pad

We will begin by looking at the one-time pad, and we explain why theoretically it is unbreakable, but in practice is it not as secure as it seems.

## 5.1.1 The basic idea

The Vigenère cipher in the previous section took the idea of encrypting letters by something different each time, and so normal cryptanalysis (such as the likelihood of “E” appearing the most often) was ineffective. Unfortunately, as we saw, because there is a repetitive keyword, once the length of the keyword is guessed, the cipher isn’t that hard to crack.

The *one-time* pad utilises a “keyword” that is as least as long as the length of the message. This “keyword” is chosen from a totally random collection set of letters, and so won’t in general represent an English phrase, so it might look like

“DHBWEHJELJFWNHQKOLRYWEQOIJSACBNNJKCFDLJFHWEUF”

Once a “keyword” is used once it is thrown away, never to be used again (hence the use of the name “one-time”).

The idea is that each party has a list of random keywords, in actual fact historically these were written on pads of paper (hence the name “pad”). The sender would encrypt the message using some randomly chosen keyword in their pad, inform the recipient of which one they used, and decryption was then easy. Then the keyword would be destroyed, never to be used again.

As long as the pads are destroyed as soon as they are used, and so never used again, the code is unbreakable, as no-one knows the “keyword” and can’t ever find it out as it has been destroyed.

Don’t worry if this sounds confusing, we will go on to explore it in much more detail in the next few pages!

### 5.1.2 Alice and Bob

You will find that in cryptographical literature, the sender and intended recipient of a message are almost always referred to as “Alice” and “Bob” (for no other particular reason than they are names beginning with A and B). We will use this convention from now on.

### 5.1.3 What actually is a one-time pad?

Suppose Alice and Bob know that they will want to communicate. In advance of the start of communications, they produce between them a pad containing many sheets of paper, each with a totally random collection of letters on them (we’ll discuss how they might do this this later). They make one copy, so Alice and Bob each have an identical pad of paper. Remember, these identical pads are made up of pages which have a long random collection of letters on them. The number of letters should be chosen to be big enough to be longer than any message they are ever likely to send.

The easiest way, is for the first message, they will use the first sheet of paper. For the second message, use the second sheet. And so on. For added security, they might agree a different order but it doesn’t make much difference – no one else has the pad, there are only two in existence and Alice and Bob have those.

I’ll give you a sample pad in the lecture. This is much larger than the pads actually were, just for ease of reading, the actual pads were tiny (although with many more pages) so the pad could be hidden and the pages could be easily destroyed – the following picture shows the size of a real one that was used.



So, suppose Alice and Bob both have the pad that I have given you (Pencils). How are they going to use this to communicate?

### 5.1.4 Modular arithmetic, numbers and letters

Remember all that modular arithmetic we did? Let’s put it to use!

We will label the letters by the numbers 0 to 25. 0 corresponds to “A”, 1 corresponds to “B”, 2 corresponds to “C”, all the way up to 25 corresponding to “Z”. So like this:

|  |  |  |  |
| --- | --- | --- | --- |
| A | 0 | N | 13 |
| B | 1 | O | 14 |
| C | 2 | P | 15 |
| D | 3 | Q | 16 |
| E | 4 | R | 17 |
| F | 5 | S | 18 |
| G | 6 | T | 19 |
| H | 7 | U | 20 |
| I | 8 | V | 21 |
| J | 9 | W | 22 |
| K | 10 | X | 23 |
| L | 11 | Y | 24 |
| M | 12 | Z | 25 |

Each letter corresponds to a number which will be our shift. So for example, a “G” corresponds to a shift of 6. Numerically, what this means is take the letter of our message, write down its number, add 6 to it (mod 26), and write down the letter corresponding to the number we obtain. Let’s see this by example.

### 5.1.5 Encrypting using the one-time pad

Having both got their copy of the pad (Pencils), Alice and Bob start communicating. Alice wants to send the message “MEET ME AT THE CAFE TONIGHT”. We’ll ignore spaces (it makes cracking much more difficult), so the text she wants to send is “MEETMEATTHECAFETONIGHT”

We’ll use the pad I have given you. She takes her copy of the pad. It’s the first message, so use the text on Page 1. We take as many letters as we need to match the message length, which is 22.

It means our text, and our “keyword” is

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M | E | E | T | M | E | A | T | T | H | E | C | A | F | E | T | O | N | I | G | H | T |
| A | P | E | N | C | I | L | I | S | A | W | R | I | T | I | N | G | I | M | P | L | E |

So Alice writes this table down. Next, convert the letters into their number equivalent.

We get

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 4 | 4 | 19 | 12 | 4 | 0 | 19 | 19 | 7 | 4 | 2 | 0 | 5 | 4 | 19 | 14 | 13 | 8 | 6 | 7 | 19 |
| 0 | 15 | 4 | 13 | 2 | 8 | 11 | 8 | 18 | 0 | 22 | 17 | 8 | 19 | 8 | 13 | 6 | 8 | 12 | 15 | 11 | 4 |

since M corresponds to 12, E corresponds to 4, and so on.

Now do the modular arithmetic. We add together the two rows (mod 26). So for example, we start with the first column and do 12 + 0 (mod 26) which is 12. Then 4 + 15 (mod 26) is 19. And so on. We get

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 19 | 8 | 6 | 14 | 12 | 11 | 1 | 11 | 7 | 0 | 19 | 8 | 24 | 12 | 5 | 20 | 21 | 20 | 21 | 18 | 23 |

Now convert these into the corresponding letters:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M | T | I | G | O | M | L | B | L | H | A | T | I | Y | M | F | U | V | U | V | S | X |

and so Alice sends the message “MTIGOMLBLHATIYMFUVUVSX”

Then, **and this is important**, she discards the first page and destroys it (maybe she shreds it or sets fire to it, or eats it!). This means the only copy of the first page is now with Bob, there were only two to start with, and Alice just destroyed hers.

### 5.1.6 Decrypting using the one-time pad

Bob gets the message and needs to work it out. He takes his pad and writes the coded message on top of the same “keyword” which he reads from his pad (remember, Alice and Bob’s pads are identical).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M | T | I | G | O | M | L | B | L | H | A | T | I | Y | M | F | U | V | U | V | S | X |
| A | P | E | N | C | I | L | I | S | A | W | R | I | T | I | N | G | I | M | P | L | E |

Again, he converts to numbers:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 19 | 8 | 6 | 14 | 12 | 11 | 1 | 11 | 7 | 0 | 19 | 8 | 24 | 12 | 5 | 20 | 21 | 20 | 21 | 18 | 23 |
| 0 | 15 | 4 | 13 | 2 | 8 | 11 | 8 | 18 | 0 | 22 | 17 | 8 | 19 | 8 | 13 | 6 | 8 | 12 | 15 | 11 | 4 |

But instead of adding like Alice did, he does the opposite, which is subtracting. So he subtracts the rows (mod 26). For example, he starts with 13 – 1 = 12 (mod 26), then 7 – 3 = 4 (mod 26). He gets

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 4 | 4 | 19 | 12 | 4 | 0 | 19 | 19 | 7 | 4 | 2 | 0 | 5 | 4 | 19 | 14 | 13 | 8 | 6 | 7 | 19 |

When checking this, don’t forget that things like 1 - 8, which is -7, are equivalent to 19 (mod 26), since -7 = (-1) \* 26 +19.

Finally, convert into letters and, hey presto, we get

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M | E | E | T | M | E | A | T | T | H | E | C | A | F | E | T | O | N | I | G | H | T |

Bob now destroys his page. There are no copies now existing of the “keyword” (Alice and Bob have destroyed the only two copies) and so no-one in the world knows the keyword. Perfect! Or is it?

### 5.1.7 The perfect code

Theoretically, this code is totally unbreakable. An interceptor doesn’t know the keyword, only Alice and Bob do, and it gets destroyed as soon as it is used. The interceptor therefore has no idea at all what the keyword is. This is not like the Vigenère cipher, when he might be able to guess the keyword length and gain some information.

Any combination of letters for the keyword (we chose them at random) is potentially possible. It’s perfectly possible to “guess” a keyword that produces a readable English sentence. But **all** sentences of that length are equally possible. Have a go at the tutorial exercises to see an example.

It was proved to be “theoretically unbreakable” by Claude Shannon in the 1940s. So why don’t we all use it?

### 5.1.8 Weaknesses of the one-time pad

In practice, the one-time pad has several weaknesses.

• Only Alice and Bob have the pad. But what happens if someone steals it? How do we make sure that no-one interrupts them while they create the pad? If someone does get a copy of the pad, then it’s easy for them to decrypt.

• Suppose one message got lost in transmission. Suddenly Alice and Bob are working on different sheets. Alice sent a message and destroyed a page. But Bob never got it, so he didn’t throw away his page. So they are on different pages now.

• Each “keyword” **must** only be used once. If even one page is ever used again, hacking is relatively straightforward using pattern-spotting techniques. We’ll see this shortly.

• Randomness. How do you create random letters? Have a look at the pad I gave you. I created this by just randomly pressing buttons on my keyboard for a while. But is this *truly* random? Look at the first line, the string “WEIUF” appears twice. Page 2 is even worse. That wasn’t intended, it’s just how my fingers worked. Also the keyboard letters aren’t equally distributed, a human tendency is to stray away from Q or P or M, for example, as they don’t have letters all around them, Q is next to the “tab” button, M is next to the comma button. Try it for yourself, sit at a computer and try and type away. Do you really feel that you are generating a totally random sequence? The question of how to generate a truly random sequence is tricky. Computers don’t work randomly, when you generate “random numbers” on a computer they are actually subject to a complex formula based on a “seed” number, which only appear random. Generating absolutely truly random sequences is very, very difficult.

• Fundamentally, how do you deal with the practicalities of creating a pad that only Alice and Bob have a copy of, and where the pages can be securely destroyed forever?

### 5.1.9 Historical use of the one-time pad

The one-time pad was used fairly extensively before the Second World War. The pad was often of the size as illustrated on the picture earlier, much smaller than your palm. The pages would often be made out of a combustible paper, so that when you had used them, they were easy to totally destroy.

One of the most famous examples of the one-time pad was the Venona project, which you can research on for the tutorial exercises. The key of the breaking of this, was that due to the amount of work in producing keywords, pages were not completely destroyed and were later re-used. That fundamental mistake renders the one-time pad almost as easy to crack as such ciphers as the Vigenère cipher.

### 5.1.10 Tutorial exercises

You might like to do these exercises in pairs, one of you being Alice and one Bob.

Q1. In the example we did before, Alice sent the message “MEETMEATTHECAFETONIGHT”. That used Page 1 of our pad. Now, Bob wants to reply with “ICANNOTMEETTONIGHT”. Encrypt this using Page 2 of your pad, and then check that decrypting it works.

Q2. Try, with friends, your own messages using the other pages. Choose whatever you like. Can you encrypt and decrypt them?

Q3. The strength of the one-time pad is that an interceptor has no idea what the keyword is and guessing is no help because every combination of letters is equally likely. Try to crack the same message “NHYYSLWXBBJIWJMNTOEKQG” as we did before, guessing the keyword “FAYDOLJXMKZGIQYAHQXGQD”. This also comes out with an English sentence. Can you come up with different keywords that also give an English sentence?

Q4. Do a little research into the Venona project. Who was involved, and what mistakes were involved that led to the cracking? There is no need to write more than about half a page but give the main details and any interesting facts you discover.

## 5.2 Stream ciphers

The one-time pad influenced the idea of what is known as a *stream cipher*.

### 5.2.1 Definition and pseudorandomness

A stream cipher is one in which, just as with the Vigenère and one-time pad ciphers, we encrypt each letter one-by-one, and change the encryption after each letter (recall that the Caesar cipher does not change the encryption).

We discussed above how, although the one-time pad is theoretically perfect, in practice it is difficult to implement and the generation of very long, totally random pads is not an easy matter.

In a stream cipher, we usually choose a key of a reasonably large size (maybe hundreds of characters long) and uses this to generate a *pseudorandom* key which we then use. What does pseudo-random mean? It means a sequence that *looks* totally random but actually is subject to a highly complex formula.

The one-time pad I gave you is pseudorandom. It looks totally random, but actually when we looked closer, the technique of me just pressing what I thought to be random keypresses wasn’t *totally* random. Human nature forced me to occasionally repeat, or omit some letters.

Here’s a way of choosing random letters. Take 26 balls each with a different letter on them (or you could use Scrabble tiles). Put them in a bag and shake them up, then pick a ball from the bag. Write that down, and replace it. Continue for as long as you need.

This is fairly cumbersome and would take ages to produce a long keyword, but is this actually random?

It seems so, but I would suggest not. What if one of the balls/tiles got stuck in the corner of the bag and we never took it? What if one ball/tile got chosen a few times and the sweat from our hands made it sticky, so we were more likely to feel it next time? It’s about as close to random as you can get, but is it *truly* random?

Can a computer produce random sequences? Well, it only has a finite memory, so eventually it is going to start repeating itself. True randomness is a very, very, difficult concept to understand.

### 5.2.2 Security and synchronisation

The sequences and keywords generated in this “pseudorandom” fashion would appear totally random to almost any of us. But the security problem is they are not *totally* random. There might be a small pattern or feature, something absolutely tiny, that an expert could spot. Yes, it might take months of intensive work, but you *might* be able to crack it if you were really an expert and had a lot of computing power. When dealing with very important secrets (such as in the military) can we afford this?

There’s another interesting point to consider about stream ciphers. Remember that they encrypt each letter one by one, according to the “random” keyword. Think back to the one-time pad. If Alice sent a message and, for whatever reason it wasn’t delivered, she has used her page of the pad but Bob hasn’t. The two pads are now “out of sync” and the future messages are indecipherable because you aren’t using the right pad. To take it to an even more basic level, if even one letter of a message is “lost” somehow, then it sets the two parties using different parts of the keywords and again messages are indecipherable.

Its not particularly unlikely for parts of a message to be corrupted or undelivered. Have you have ever had an email message bounce? Or a telephone conversation suffer from a bit of interference or background noise and means you don’t quite hear what your friend said? No communication is 100% secure.

One way to get around this is to send “markers” after a certain number of letters, to ensure that the two parties are back in synchronisation at each marker. But there is the susceptibility for a malicious attacker to intercept a message and add or remove a single letter (e.g. by generating some noise on the line) and mess up a good chunk of the message, up to the next marker at least.

We’ll talk a bit more about this sort of thing when we discuss attacks later on.

### 5.2.3 Tutorial exercise

Try to think of the most random way you can, to pick a sequence of letters or numbers that could be used in a stream cipher. Describe your method. How random do you think it is? Does it have any flaws? There are no right and wrong answers to this!

## 5.3 Block ciphers

We will now look at an alternative concept to stream ciphering. Much of this is currently in use, which we discuss when we look at the Data Encryption Standard and Advanced Encryption Standard in the next section.

### 5.3.1 Definition of block ciphers

A *block cipher* is one, which in contrast to a stream cipher which operates on each letter one-by-one, operates on *blocks* of characters. Typical sizes are 8 or 16 characters. So for example, the block cipher might work on 16 characters. So it takes in a text of length 16, and uses some algorithm to convert this to another block of text of length 16.

The transformation (as defined by the key) used is unvarying, that is it acts the same way on each block. This is in contrast to many stream ciphers, where the operation changes after each character. The distinction between block and stream ciphers can become somewhat blurred. For example, if the way the block of 16 characters is encrypted is by using a symbol-by-symbol method then this effectively reduces to a stream cipher.

A block cipher uses a symmetric key, that is decoding is done by using the same key (or obtained from it trivially) as the key for encoding. This has the advantage of speed and ease of decryption, but the disadvantages of the natural problems of keeping the key secret, changing it regularly, and distributing it wisely.

### 5.3.2 Arbitrary string lengths

Let’s assume we are using a block cipher on 16 characters. Of course, not every message is of length 16! A few are, but some are shorter, and many are longer.

For messages longer than 16 characters, split the message up into the first 16 characters, then the next 16 characters, and so on. So a message of 64 characters is split into four messages (characters 1-16, then characters 17-32, then characters 33-48, then characters 49-64). Each message is then encoded using the cipher. To further add security, the splitting-up is often done using a so-called *mode of operation*. This is really beyond the scope of this module but we will briefly discuss it if we have time.

But we still have a problem. When we split up like this, our last “block” is likely to be less than 16 characters long. But block ciphers work on a fixed length, so we need to “make” the string 16 characters long for our cipher to work. Similarly if we have a very short message, less than 16 characters long, then again we need to “make” it 16 characters long for our cipher to work.

This is usually done by *padding*. For example, we simply add enough of an “extra” symbol to the end to make the length up to 16.

So for example the string

“ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXZ”

is 52 characters long, and I want it to be 64 characters long so that it’s a multiple of 16 (so can be written as a collection of strings 16 characters long). If I used the padding symbol #, this would become

“ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXZ############”

The length of this message is now 64 as I’ve added in an extra 12 # symbols, which makes it a nice multiple of 16 so can be split into convenient blocks of 16, namely:

“ABCDEFGHIJKLMNOP”

“QRSTUVWXYZABCDEF”

“GHIJKLMNOPQRSTUV”

“WXYZ############”

so I can use my block cipher on 16 characters.

This is fine, but you have to be careful to choose the symbol wisely. Here’s a simple example. I’ve decided, in my infinite wisdom, that we don’t usually have numbers in my messages, but I do use punctuation and spaces, so I’ll choose a number as my padding symbol, let’s say 5.

I want to send this message:

I HAVE THE DOCUMENTS. MEET ME AT THE MEXICAN RESTAURANT AT 10.

If you count (including the spaces and punctuation) this is 62 characters long. So I’ll pad it with the symbol 5 to make it 64 characters long. I get:

I HAVE THE DOCUMENTS. MEET ME AT THE MEXICAN RESTAURANT AT 10.55

See the problem? This is a particularly stupid example, but it’s important for illustrative purposes.

Padding with a single symbol also has the disadvantage that it becomes easier to crack and work out the length of the actual message. More complex schemes are used in modern systems, which pad with seemingly random characters. But again you must be very careful. What happens if, by chance, the random characters actually make a word?

Some of you might be aware of the concept of the “monkeys and the typewriters”. The concept behind this, is that if you have infinitely many monkeys randomly pressing typewriter keys, eventually one of them will type any string you care to think of, so at some point, one will, by pure fluke, type the complete works of Shakespeare. It’s almost impossibly unlikely, that the random choice of letters they chose made such a long string, but *it is possible*.

So for example, take a similar message to above

I HAVE THE DOCUMENTS. MEET ME AT THE RESTAURANT AT 10.00

This needs padding by 8 characters. We’d like them to be random, so for example we might get this:

I HAVE THE DOCUMENTS. MEET ME AT THE RESTAURANT AT 10.00DQIWRZST

It’s obvious to the decoder that the last few letters are just padding and they understand the message.

But suppose that, chosen completely randomly, by some method, I chose the random letters T, O, M, O, R, R, O, W. How likely is this? Well, not very likely, but possible. I get the message:

I HAVE THE DOCUMENTS. MEET ME AT THE RESTAURANT AT 10.00TOMORROW

Again, a problem. Or I might choose M, O, N, Q, W, E, Z, X

I HAVE THE DOCUMENTS. MEET ME AT THE RESTAURANT AT 10.00MONQWEZX

Is the MON supposed to mean Monday with then some padding symbols after it? Or is it padding itself? How does the person know?

So the skill to choose padding symbols wisely is very important.

### 5.3.3 Binary numbers, bits and characters

Another issue arises when you consider coding in the electronic age. Now everything is done by computers. Above, we always talked about characters, letters, symbols – that’s how coding was always done, it’s the language we understand! But what is a “letter” to a computer?

Computers are *binary* devices. This means that the things they understand are lists of 0’s and 1’s. When you give them commands, letters, sentences, etc, then the computer converts that into binary (0s and 1s) so it can understand it (a little like you would do if asked to translate a document from another language – you’d convert it into a language you understand)

Most computers work on reading in symbols that the user types, from a potential “alphabet” of 256 characters. This seems a rather strange number but we’ll see the reason shortly. Also, it seems quite a lot. After all, English has only 26 letters in its alphabet, what else would they type? Well, what about lower case letters? That’s another 26 characters immediately. And then punctuation symbols, mathematical symbols used in everyday life like +, -, \*, /, % etc, spaces, and all sorts of other things?

Remember that computers use binary. A *bit* is one single binary symbol, so either 0 or 1. In computer terminology, a *byte* is a collection of 8 bits and these are usually processed as blocks by the computer. So examples of *bytes* are 10100101 or 00010010 or 00110101 – you get the idea.

How many different possible “bytes” are there? Well, the first symbol can either be 0 or 1. So there are two possibilities for the first symbol. For each of these, there are then two possibilities for the second symbol (0 or 1), so in total there are 2\*2 = 4 possibilities for the first two symbols (either 00, 01, 10 or 11). Then the next symbol can either be 1 or 0. There are 2\*2\*2 = 8 possibilities for the first three symbols (000, 001, 010, 011, 100, 101, 110, 111) and so on, 2\*2\*2\*2 = 24 = 16 possibilities for the first 4 symbols, 2\*2\*2\*2\*2 = 25 = 32 possibilities for the first 5 symbols, 2\*2\*2\*2\*2\*2 = 26 = 64 possibilities for the first 6 symbols, 2\*2\*2\*2\*2\*2\*2 = 27 = 128 possibilities for the first 7 symbols and finally 2\*2\*2\*2\*2\*2\*2\*2 = 28 = 256 possibilities for the whole byte. That’s where the 256 comes from!

The common ASCII code for computers assigns each character in our alphabet (remember this includes upper/lower case letters, punctuation symbols etc) to a number between 0 and 255 (so 256 numbers in total). Usually ‘A’ is assigned the number 65, ‘B’ 66 and so on, with punctuation/special characters before and after – though this is purely conventional. This number is converted to a byte for the computer to use.

Let’s illustrate this by a couple of examples. What symbol corresponds to the binary code 01001101? If you don’t already know, you need to know how binary numbers actually work! *Skip this section only if you are absolutely confident with binary numbers!!!!!*

There’s a simple algorithm for you to follow. With a bit of practice, this will become second nature.

* Start by writing a table with powers of 2 on the top row, *starting from the right-hand side*. So we write 1, then 2, then 4, then 8, then 16, and so on, like this:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

* Now write the code directly underneath this table. In our example the code is 01001101 so we write exactly that, giving us this:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

Now go through the whole table, multiplying the top number by the bottom number in each column and adding all these products together. We get

128\*0 + 64\*1 + 32\*0 + 16\*0 + 8\*1 + 4\*1 + 2\*0 + 1\*1

= 0 + 64 + 0 + 0 + 8 + 4 + 0 + 1

= 77

77 corresponds to the symbol ‘M’ in the usual ASCII alphabet and so this is the binary code for the letter ‘M’.

In actual fact, there’s a clear shortcut for binary numbers, which is just to add together those numbers that have a 1 underneath them (so we get 64 + 8 + 4 + 1 = 77) but I’ve given the full details to try to help you see that binary numbers aren’t as scary as they might seem. Let me explain.

Suppose I asked you to give me a random 4-digit number. No idea what you might choose, but let’s say it’s 5372. Say that number out loud - “five thousand, three hundred and seventy-two”. You’re counting the number of thousands, the number of hundreds, the number of tens, and the number of ones. Let me draw a table like before, but I’ll use powers of 10, not powers of 2, and I’ll only include 4 columns as our number is only 4 digits long.

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 | 100 | 10 | 1 |
| 5 | 3 | 7 | 2 |

See? Applying the same as before, we have 5\*1000 + 3\*100 + 7\*10 + 2\*1 = 5372. You’ve been doing this sort of thing automatically in your head since primary school - all we are doing with binary numbers is using powers of 2, rather than powers of 10!

How do we do it the other way round? What is the binary code corresponding to the symbol ‘G’? Looking up, this has ASCII code 71. How do we write 71 in binary? Again take your list of powers of 2 like before, leave the row below blank for now:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

Now follow an algorithm like this:

• The basic idea is that we want to make 71 from a combination of these powers of 2

• *Start from the left hand side*

* Look at the first number in that table, its 128. 128 is too big, we can’t add 128 to the other powers to make 71! So we don’t want any 128s. Write 0 to indicate we don’t want any 128s.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 |  |  |  |  |  |  |  |

* 64 is smaller than 71, so we want a 64 (if you think about it for a while, you’ll realise that if we have no 64s, we can’t possibly make 71 from what’s left as they only add up to 63 even if we chose them all) – but in any case just follow that rule – if the power is smaller than the number we want, then we want 1 of it

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 |  |  |  |  |  |  |

* Keep working left to right, deciding whether we need the power or not.

We’re trying to make 71 and we’ve already got 64, so we need another 7. Do we need a 32? No, it’s too big, so we want no 32s.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 |  |  |  |  |  |

* We want to make 7. Do we need a 16? No, it’s too big, so we want no 16s

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 |  |  |  |  |

* We want to make 7. Do we need a 8? No, it’s too big, so we want no 8s

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 | 0 |  |  |  |

* We want to make 7. Looking at the next column, 4 is smaller than 7 so we need a 4.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 |  |  |

* Now we just need to make another 3 (we already have 64 and 4 giving us 68, so just another 3 needed to make 71).
* 2 is smaller than 3, so we want a 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 |  |

* Finally we want just 1 more, we already have 70. So we need a 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

Hence, reading along the final row, our binary equivalent to the letter ‘G’ (the binary number corresponding to 71) is 01000110. You can check this works (64 + 4 + 2 + 1 = 71).

This seems complicated as a first example, but after a few goes, it will become much easier - just follow the basic rules given.

### 5.3.4 Tutorial exercises

Q1. On your computer, open up Word, or some text editor. Type away as randomly as you can for a page or two. Now go back and look at what you wrote. Is there anywhere in there that you can see 1-letter words? 2-letter words? 3-letter words? And so on? Chances are you typed in some small words completely by accident. Now imagine typing like that for years. Is it likely you’ll eventually type in by chance some quite long words?

Q2. Find from the web some estimates as to the number of emails sent each day around the world.

Q3. (*Ask me if you don’t know any probability*). Let’s suppose our padding symbols were to be chosen randomly from the letters A-Z. So the probability of choosing a particular letter each time is 1/26. What is the probability I choose T, O, M, O, R, R, O, W as a sequence of 8 letters? What is the probability I chose M, O, N and then any other 5 letters? If we all used this padding mechanism, and given your answer to question 2, and the fact that email is only a small proportion of all communications, would these be a realistic likelihood to ever occur?

Q4. What numbers do the following bytes represent?

(i) 10010100 (ii) 01011011 (iii) 00000000 (iv) 111111111

Q5. Using standard ASCII codes, so ‘A’ is 65, ‘B’ is 66, up to ‘Z’ is 91, convert the message “BYE” into binary (*just take each letter, work out its ASCII number, and convert that into binary. Then just write the bytes next to each other, so overall it will be 24 bits (0s and 1s) long since there are 3 letters, so 3 bytes, each 8 bits long).*

Q6. Decode the binary string:

“010010000100010101001100010100000100110101000101”

(*split it up into blocks of 8 symbols and decode each one to get the ASCII number, then write down the corresponding letter)*

## 5.4 Data Encryption Standard and Advanced Encryption Standard

One of the most widely-used and popular block ciphers was the Data Encryption Standard, superseded by the Advanced Encryption Standard in the early part of this millennium. In this section, we will look at the history and basic ideas behind these standards.

### 5.4.1 Data Encryption Standard

The Data Encryption Standard is often cited as the cipher that motivated the interest in cryptography and propelled the ideas of cryptanalysis, which we will discuss in more detail later.

The cipher was first developed in the early 1970s, and was adopted as an official standard in 1976. The intention from the US security agencies was to adopt a secure, common system, to be used throughout the US. Work between the US National Security Agency (NSA) and telecommunications giant IBM led to the formal definition of the cipher and its eventual adoption worldwide. Being described as “secure” it naturally motivated a deeper interest in cryptography as researchers took on the challenge to break it.

The Data Encryption Standard is theoretically a 64-bit block cipher (so uses a key of length 64 bits and each message is split into blocks of 64 bits). However, the final 8 bits of the key are used as *parity* (or checking) bits to ensure the integrity of the message. Hence, effectively the key length is only 56 bits (so corresponding to 7 characters).

This is a relatively short key length and eventually led to it being cracked. Why was it so short? The official view is that it was deemed sufficient for all purposes at the time by both IBM and the NSA. There are widespread suggestions that in actual fact, the NSA wanted a shorter key length to be adopted as an international standard, as they felt they had the computing power to be able to break the 56-bit in advance of the rest of the world and so put themselves at an advantage if it was adopted worldwide. These suggestions remain unproved and are only conjectures.

We will give the basic idea behind its operation without going into too much detail. Remember that this is a 64-bit cipher, and so takes in a block of length 64 (so we split our message into blocks of 64 bits, padding the final block if necessary, as discussed above).

We start by permuting the bits under some random permutation (that is, we move the bits around) called the *initial permutation*. When finished, we apply the inverse of this, called the *final permutation,* to get everything back to normal. This isn’t really an essential process but was done to ease the process of implementing the cipher on the hardware available.

In between, we apply a sequence of 16 identical *rounds* which act on the string of bits we currently have. Why 16? The real reason is that it’s simply “enough” rounds to make the cipher secure. Only a few rounds are easy to crack, but having more than 16 doesn’t really help to make the cipher any harder to crack.

After the original permutation, we split the message into two halves (so 32 bits each) and on each round, we act on only the first half and then swap the halves over. The fundamental idea behind the action involves combining together the message bits and the key using an *exclusive or (XOR)* operation. What does this mean?

The XOR operation takes in two bits, and returns the value 1 if one of them is 1 and the other 0. If they are both 0, or both 1, it returns 0. The name “exclusive or” refers to the fact that it returns 1 (corresponding to “true” in computer programming, if either one **or** the other of the bits is 1 (true), but **excludes** the possibility that *both* are true). The following table summarises the XOR operation.

|  |  |  |
| --- | --- | --- |
| Input 1 | Input 2 | **Output** |
| 0 | 0 | **0** |
| 0 | 1 | **1** |
| 1 | 0 | **1** |
| 1 | 1 | **0** |

To operate on a string of bits, take the original string, write the keyword in bits underneath it, and take the XOR of each pair of bits, one-by-one. So, for example (I’ve only used 16 bits to fit nicely on the page), suppose we have initial string 1001011100010101 and keyword 0011010100011001.

Writing them in a table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |

And then taking the XOR of the two bits in each column we get:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

The advantage of using XOR is that if you were to apply it again (using the same keyword) then you get the original message back again. So decryption is easy – this is a feature of “symmetric” algorithms where the same key is used for encryption and decryption.

The overall operation of the Data Encryption Standard is summarised in the diagram on page 68. This diagram illustrates the fact the original message is split into two halves, and one half is operated on at each stage. In each round, the whole scheme follows the *Feistel* scheme in what is known as an “F” (for Feistel) box. Without giving the full details, at each stage (in the “F” box) we:

• Expand the 32-bit half-block to 48 bits (standard algorithms exist to do this, for example duplicate every alternate bit)

• Derive a 48-bit key from the main 64-bit key using some algorithm (there will be a different key at each round), and take the XOR of these two 48-bit strings

• Split the answer into 8 mini-blocks of 6 bits, and apply a “substitution” to each mini-block to convert each one into a block of 4 bits (using some sort of fixed encryption scheme, for example looking up in a table a conversion for each possible mini-block)

• This leaves us with a 32 bit string, which we can apply a fixed permutation to and then get our final coded string. Note this is 32 bits long, just as the original half-block.

If this sounds complicated, even to explain at a basic level, that’s because it is :) Remember, this cipher was used on a global scale for the most sensitive of data. But if you know the key and the algorithms, it’s a practical cipher because decryption is easy. Of course, keeping the key and algorithms secret is a more difficult task.

The most important aspect of the cipher is the substitution stage, where the mini-blocks of length 6 are converted into blocks of length 4. This part of the process is the fundamental aspect of what makes the cipher secure.

Since the US Government had described it as “secure” it was a natural challenge to try and crack it. Cryptography as a subject really grew from here, as researchers, hackers and mathematicians worked towards cracking it and trying to prove it wasn’t quite as secure as claimed. Theoretical attacks were put forward, although not practically implementable. However, in 1998 a machine “Deep Crack” was built by the Electronic Frontier Foundation which was able to crack the DES in less than 2 days. It’s an entirely brute-force algorithm, it just keeps trying all possible keys, but computing power had grown to the level that this was actually feasible. The machine cost around $250,000 to build – a pittance for large corporations and governments.

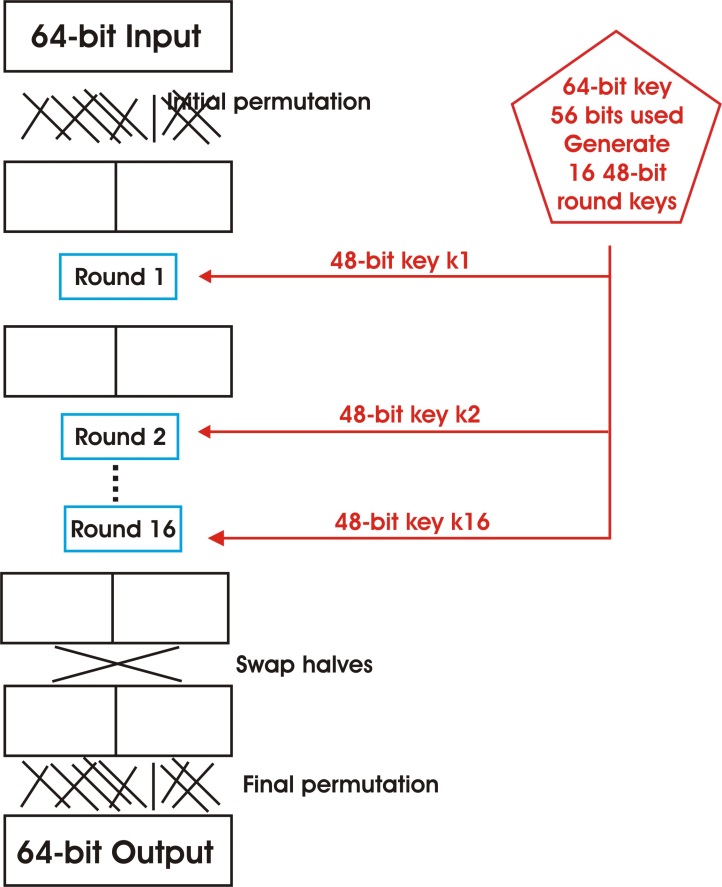


Figure 1 - DES

The fact is, that although 56 bits was big enough when the cipher was invented in the 1970s, computing power has grown to such an extent that it was now practically breakable and 56 bits is just too small. You all know how computing power has grown – remember there wasn’t even an internet when you were born, now can you imagine life without it? When the cipher was invented, the cost and time to create a machine to try every possible key was unimaginable and so it was considered secure. By the 1990s, it wasn’t just imaginable, it was really practical.

### 5.4.2 Advanced Encryption Standard

With the DES looking insecure, it was time for a new algorithm. After Deep Crack’s breaking of DES, the US invited cryptographers to submit a new cipher to supersede DES. The winners were two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who created the cipher they called “Rijndael” (pronounced something like “Rhine-dahl”) which was adopted as the new national standard in 2001. This is called the Advanced Encryption Standard and is in use for classified material worldwide.

A detailed description of this cipher is beyond the scope of this module but feel free to do any research and find out what you can!

It uses blocks of 128 bits and keys of length either 128, 192 or 256. This is beyond what any machine can “brute-force” attack (i.e. by trying every key) under current computing power. Moore’s Law (which is not a law at all, just a theory), is that computing power doubles roughly every two years. Increasing the key length by 1 doubles the number of possible keys. We’ve only just cracked 56 bits, so we are looking at many decades before we can brute-force an attack on AES.

There are concerns about it. It has a deep mathematical structure, and there are fears that the logic behind it may prove to be its downfall as it may prove to be logically crackable. And eventually it will have to be superseded as computing power does grow to a level where it can be practically attacked. But at the moment, it can be considered unbreakable under modern computing power.

### 5.4.3 Tutorial exercises

Q1. Take the XOR of the input string “1011010101111110” and key string “0101100000101010”. Then take the XOR of your answer with the key string again. You should get the original string back again.

Q2. Research and find out what you can about Deep Crack. See if you can find any pictures of it, or any interesting facts.

Q3. Write down a table showing the number of possible keys of bit length 1, 2, 3, 4, and 5. Show that, in general, increasing the key length by just 1 bit has the effect of doubling the number of possible keys (*this is easy* - *consider all possible keys of a given length. Increasing the length by 1 means we can add either a 0 or 1 to the end of each. So how many are there now?)*

Q4. If Moore’s Law is actually valid, so computing power doubles every 2 years, and the 56-bit DES code was broken by brute-force in 1998, when can we expect the 128-bit AES to be broken by brute-force?

# Section 6 – Public Key Cryptography

We will move on now to consider another approach to cryptography. This takes a different tack to the approach that we have followed before in that the algorithms are no longer symmetric – that is, the same key is not used for decrypting as well as encrypting. Most such algorithms rely on problems that are easy to formulate but practically impossible to solve – this is where the number theory we talked about at the start of this module really comes into play. We will illustrate the concept by the well-known RSA algorithm.

## 6.1 History and underlying ideas

The main weakness behind all the algorithms we have talked about so far, is that they require both the sender and the recipient to use an identical key. This sounds secure, but has the logistical problem of actually creating the key and making sure both have an identical copy at the start. This has to be done by some form of distribution – you could post the key, but what if a corrupt messenger “borrows” it and copies it?

Public-key cryptography works on the idea that everyone has their own secret key and they keep it totally to themselves. It will become a lot clearer when we give an example, so this section is deliberately brief.

The idea behind public-key cryptography, which is to have a problem that is easy to formulate but hard to solve, has been known mathematically for centuries – for example, it’s very easy to multiply two numbers together, just do long multiplication, but very hard to do the opposite – that is, to factorise a number – for a big number, there are so many possible factors to check.

The first published public-key algorithm was by Diffie and Hellman in 1976, although it is generally accepted that researchers at GCHQ in the UK were using the concept a couple of years earlier. The famous RSA algorithm, that we will discuss next, was published in 1978. This is still in many ways a developing subject and is intrinsically linked to pure mathematics – in the unlikely event that these problems we “believe to be hard” are eventually “solved” (so easy algorithms are found), then breaking these codes is easy.

To give a very informal idea and illustration about what public-key cryptography means, imagine sending a secret between two people (Alice and Bob) using a padlocked box. In the systems we have used so far, Alice writes her message, puts it in the box, locks it, sends it to Bob, who unlocks it using a copy of Alice’s key and reads the message. To reply he does the same thing, they have the same key.

In public key-cryptography, Bob sends a box with an open lock to Alice but keeps the key to himself. Alice writes the message, puts it in the box, and then closes the box (like with most front doors, if you close it, it automatically locks). She sends the box to Bob, who unlocks it using his own key. Alice has never had to use a key and no-one ever to copy or transmit a key.

To reply, Alice sends her open box to Bob who follows the same procedure.

Inherently, this still has weaknesses in that the key can be compromised or “hacked” but it appears to be more secure, in that at least the key is never sent anywhere so the chances of someone just intercepting it is removed, although the chance of it being “discovered” by brute-force or any other attack still exists.

## 6.2 The RSA algorithm

It really is easier to see all of this by example so let’s look at the RSA algorithm, which we will cover in full detail.

### 6.2.1 History and background

The RSA algorithm was first described in 1977 by Ron Rivest, Adi Shamir and Len Adleman (hence the name RSA, after the initial letters of their surnames), cryptographers working at the Massachussetts Institute of Technology, and was actually registered as a patent in the US in the early 1980s. The fact that a similar algorithm had been invented in GCHQ in the UK a few years earlier, by the mathematician Clifford Cocks, only came to light in 1997 due to the secret classified nature of GCHQ’s workings.

Rivest, Shamir and Adleman were awarded the Turing prize in 2002 for their work on the algorithm – this prize is considered as the “Nobel Prize” of computer science.

Here is a picture of them from the 1970s:



It uses a *public key* and a *private key*. The private key is known **only** to the person who created it and is never distributed (so, for example, Alice and Bob don’t know each other’s private keys). The public key can be distributed – the idea is that if someone does know the public key, they still can’t decipher the message, so it can be considered as being available to anyone and it doesn’t matter if enemies know it or not.

### 6.2.2 An example

Let’s work through, in full detail, an example of using the RSA algorithm. As usual, Alice will try to send a message to Bob. The communication channel isn’t particularly secure (like the internet) so they don’t want to run the risk of having to transmit their key, so they decide to use the public-key RSA algorithm.

Don’t worry if this isn’t totally clear on a first read-through and attempt, take it absolutely step-by-step and it should become clear. The mathematics behind some of the steps is contained in the next section, so that you can hopefully get the idea without too much technicality at first.

Alice first of all converts her message into a number. This isn’t a trivial procedure but she can do this in many ways, for example, using ASCII codes, or bits, like before. It’s assumed that they’ve already discussed this and both know how to do it, and can extract the written message from the number and vice versa (like before in the tutorial exercises, when you extracted a message from a collection of bytes of ASCII codes and vice versa). The process of doing this is not really relevant to the discussion here of how RSA works, so we assume they are just sending numbers and they know how the numbers relate to the actual message.

Now we need to choose keys. This is where the mathematics we did at the start of this module comes into play.

• Bob begins by choosing two, different, very large prime numbers *p* and *q* at random. He may get these from a list of primes, or using a computer, but they should be random and not connected or dependent on each other in some obvious way. In this example, I will use the *tiny* prime numbers *p =* 61 and *q =* 53 so you can see the idea – numbers actually used in the real world tend to be 1024 or 2048 bits long. He doesn’t tell anyone his choice of numbers, so they are private.

• He then works out the product *n* = *p \* q*, and sends this number *n* to Alice. He doesn’t worry too much if it gets intercepted, it doesn’t really matter who knows it, so this number can be public. Hence in this case, he calculates *n = 61 \* 53 = 3233*. Now, if someone finds out this number by intercepting the message, all they have to go on is the number 3233. They have to try and work out the factors *p* and *q*. OK, for a number like 3233 it wouldn’t take that long, especially on a computer, but for the sort of numbers used, factorising them is virtually impossible and certainly impractical under modern computing power – *remember, multiplying is easy, but factorising is hard*.

Don’t forget – the number *n* can be freely available, it really doesn’t matter if anyone knows it - the numbers *p* and *q*, only Bob knows because he picked them, and they are so big that they can’t be easily worked out by factorising *n*, even by the most powerful computer.

• Next, he works out the Euler totient function *φ(n)*. Remember that this is the number of integers less than *n* which are coprime (have no factors in common) to *n*. If it’s not obvious to you, we’ll show in the next section that since *n* is the product of two primes *p* and *q*, then *φ(n) = (p – 1)(q – 1).* So in our case, *φ(n) = 60 \* 52 = 3120*

• Next, he chooses some number (public exponent) *e* which is smaller than *φ(n)* and coprime to it. It doesn’t need to actually be that large for decent security, let’s just say we choose 17. He sends this to Alice as well, again it doesn’t really matter if it’s intercepted, so it can be classed as public.

• Finally, Bob works out a number (private exponent) *d* such that *de ≡* 1 (mod *φ*(*n*)). How he actually does this is covered in the mathematical section to follow. But the main point is that you need to know *p* and *q* to do this this, because you need to work out *φ*(*n*) = *(p – 1) (q - 1)*, and only Bob knows these numbers *p* and *q*. Bob keeps this entirely to himself and doesn’t communicate it, so it remains private to him. This will be his key for decrypting Alice’s message when she sends it to him. You can show that *d* = 2753 is a suitable solution. Only he knows this key.

Bob has now done all he needs to do. He’s provided Alice with a public key *n* and the public exponent *e*, which we aren’t particularly bothered about if they are intercepted or found out by anyone else (hence called public). Over to Alice and actually sending the message now.

**Remember that at this point, the numbers *n* and *e* might well be known by a hacker who intercepted him sending them to Alice, in fact Bob doesn’t care if they are, they can be as public as they like, but the numbers *p, q,* and *d* that Bob invented and worked out are entirely private to him and he has never even tried to distribute them to anyone else, so nobody else knows them, and they can’t be worked out from *n* and *e* (at least, not with current computing power).**

So, here’s what Alice does:

• She takes her message and converts it into a number, let’s say *m* – remember Bob and Alice have already agreed how to do this beforehand. The only important thing is that *m* is smaller than the number *n* that Bob sent her – if it’s not, she’ll split the message up in the usual way, but since *n* is usually chosen to be huge, this is rarely a problem. Suppose, just as an example, the number corresponding to her message is 123.

• She works out the number *c = me (mod n)* – remember that she already knows *e* and *n* since Bob sent them and made them public, and she just obtained *m*, so this is fairly easy for her to do – for example she can use the quick-exponentiation technique we talked about before. In this case, *m = 123, e = 17, n = 3233* and *(123)17 (mod 3233)* pretty quickly works out to be 855.

• She then sends this number *c =* 855 to Bob.

• Bob can work out the original number *m* by calculating *m = cd (mod n)*. Again I’ll give the maths behind this in the next section. So in this case, he works out *8552753 (mod 3233)*, again probably using fast exponentiation, which does return the answer 123 which was Alice’s coded message.

Wait a minute, what’s the point? Why is this secure?

**The point is that the only person who can work out *cd (mod n)* is someone who knows what *d* is. And the only person who knows what *d* is, is Bob. No-one could have hacked it, because he never tried to distribute it. Even Alice didn’t know it. So no-one else can work this value out and hence read the message.**

### 6.2.3 The underlying mathematics

I deliberately omitted some of the underlying mathematics from the previous section so that you could have a fairly straightforward and clear algorithm to follow to understand RSA coding, so I have put this mathematics in a separate section.

**Don’t be daunted looking at this.** The trick of reading mathematics is to take it step by step, each step is relatively simple so if you take it slowly, you work your way through to the end.

#### Problem 1

First of all, let’s point out why the Euler totient function *φ*(*n*) is *(p – 1) (q - 1)*. There are several ways to prove this, I’ll give one, you might like to find another way. Remember that *n = p \* q* where *p* and *q* are prime, so this is its prime factorisation. So the only numbers that are not coprime to *n* must have a factor of at least one of either *p* or *q*. Well, we have all the numbers from 1 up to (but not including) *n* to consider, and so let’s eliminate the ones that aren’t coprime, which will leave us with all the coprime ones and we’re done. There are *n - 1* numbers to consider (1 up to *n – 1*, remember we don’t count *n* itself). Since *n* = *pq* then *n – 1 = pq – 1*. So we have *pq – 1* possible numbers. Of these, the numbers *p, 2p, 3p, 4p, ……, (q - 1)p* are all the multiples of *p*, that is all the numbers that have *p* as a factor. We don’t have any more because the next multiple of *p* is *qp* which is *n*, and so we’ve now gone too far, we’ve reached *n*. So, counting them, there are *(q – 1)* numbers less than *n* which have a common factor of *p* and so we need to eliminate those *(q – 1)* numbers. Similarly, there are *(p – 1)* numbers less than *n* which have a common factor of *q* (the numbers *q, 2q, 3q, 4q, ……, (p - 1)q*) and so we need to eliminate those *(p – 1)* numbers. Every other number must be coprime to *n* because the only two prime factors of *n* are *p* and *q* and we’ve already accounted for all the multiples of those – nothing else can have a factor in common. So, the total numbers less than *n* which are coprime to *n* can be worked out by:

Start with all possible numbers (we agreed there are *pq – 1* of them) Take away the multiples of *p* (we agreed there are *q – 1* of them) Take away the multiples of *q* (we agreed there are *p – 1* of them) Overall then, we have (*pq – 1*) – (*q – 1) –* (*p – 1*) coprime numbers. This works out to be *pq – 1 – q + 1 – p + 1 = pq – p – q +1* If you check by multiplying out the brackets, this is the same as *(p – 1)(q – 1)* and so we’ve finished the proof. We did indeed show *φ*(*n*) = *(p – 1) (q - 1)*.

Problem 2

Next problem, how does Bob work out *d*? Remember, he wants a number *d* such that *de ≡* 1 (mod *φ*(*n*)). I’ll use the exact same example that we did above, so we’re trying to find a number *d* such that 17*d ≡* 1 (mod 3120). **Remember,** that only Bob knows this number 3120 because only he knows *p* and *q*, and 3120 = (*p – 1)(q – 1)*, so only he can work this out.

If 17*d ≡* 1 (mod 3120) then, by definition, that means that 17*d* is one more than some multiple of 3120 (so could be 3121, 6241, etc). Hence an equivalent formulation is to solve the equation 17*d* + 3120*a* = 1 – we want the answer *d*, it doesn’t matter what *a* is.

This can be solved using Euclid’s algorithm. Hopefully you’ve seen this before, but if not, don’t worry, you don’t need to know the deep theory behind how this works, I’ll just give you a set of steps to follow. Again, there are loads of ways to actually do this, I’ll give you probably the simplest to work out by pen-and-paper but feel free to look at other implementations better suited maybe for a computer?

Start by writing a table with two rows and three columns, and fill in the first row with the numbers 1, 0, and our *φ*(*n*), which is 3120. Then fill in the second row with the numbers 0, 1, and our *e*, which is 17. So our table starts off like this:

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 3120 |
| 0 | 1 | 17 |

Now, look at the last column and work out how many times 17 divides into 3120 (the quotient). If you check, you’ll see that 3120 = 17 \* 183 + 9. So the quotient is 183 and the remainder is 9.

The rule now is to add another row to our table, and to make the numbers in this row, for each column we take the number in the row two above (the top one here), and subtract the quotient multiplied by the number in the row one above (the second row).

So, in the first column, we take 1 (the number in the top row), and subtract 183 (the quotient) multiplied by 0 (the number in the second row), that is 1 – 183 \* 0 = 1 – 0 = 1

In the second column, we do the same thing, so we take 0 (the number in the top row), and subtract 183 (the quotient) multiplied by 1 (the number in the second row), that is we do 0 – 183 \* 1 = 0 – 183 = -183

Similarly in the third column, we do 3120 – 183 \* 17, which gives us 9 to fill in in that column. Hence we now have the table

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 3120 |
| 0 | 1 | 17 |
| 1 | -183 | 9 |

What we are going to do is repeat the procedure until we get down to 1 in the last column (*if you know the theoretical concepts behind this algorithm, you’ll know that we must get down to 1 since we will always end up with the highest common factor of the two starting numbers, and we deliberately chose 17 to be coprime to 3120 and so the highest common factor is 1*).

So next time, we try to divide 9 into 17. The quotient is 1, and remainder 8, since doing the division, 17 = 9 \* 1 + 8.

To fill in the next row of our table, do exactly the same thing as we did before. The quotient this time is just 1. So, we use the second and third rows (the two rows above) take the numbers in the second row, and subtract 1 multiplied by the numbers in the third row.

Hence we get 0 – 1 \* 1 = 0 – 1 = -1 in the first column, 1 – 1 \* (-183) = 1 + 183 = 184 in the second column, and 17 – 1 \* 9 = 17 – 9 = 8 in the last column. So we have:

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 3120 |
| 0 | 1 | 17 |
| 1 | -183 | 9 |
| -1 | 184 | 8 |

Same thing, work on the last two rows and try to divide 8 into 9. The quotient is 1 and the remainder is 1, since 9 = 8 \* 1 + 1. So work out the fifth row by taking the third row and subtracting 1 (the quotient) multiplied by the fourth row.

This gives 1 - 1 \* (-1) = 1 + 1 = 2 in the first column, -183 – 1 \* (184) = -183 – 184 =

-367 in the second column, and 9 – 1 \* 8 = 9 – 8 = 1 in the third column.

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 3120 |
| 0 | 1 | 17 |
| 1 | -183 | 9 |
| -1 | 184 | 8 |
| 2 | -367 | 1 |

We’ve now finished, since we got down to 1 in the last column. We can simply (you don’t need to know why if it’s not immediately clear, so don’t worry) read off our answers from the last row.

We have 2 multiplied by the first number we started with (3120), added to -367 multiplied by the second number we started with (17) to give the answer 1. Does this work? Let’s check.

2 \* 3120 + (-367) \* 17 = 6240 – 367 \* 17 = 6240 – 6239 = 1

Thinking back to our original question, and how we formulated it, the number *d* we were looking for is what we multiply *e* (17) by, which is (-367).

Remember though, that when we are talking (mod 3120), we’d normally not use negative numbers (like we would usually say 5 (mod 7) rather than (-2) (mod 7).

(-367) ≡ 2753 (mod 3120) (you can easily check this) and so Bob’s *d* has worked out to be 2753.

I know this seems long-winded but after a bit of practice you’ll be able to do it quite quickly. Also, we could relatively easily program a computer to do it.

#### Problem 3

Remember I said that Bob can recover the message by working out *m = cd (mod n)*, which he can do since Alice sent him *c,* and *d* and *n* are private to him, he invented them! We’ll prove that this does work.

As you read through the following description, you could fill in the values of *c, m, e, n, d* that we have been using in our example, to convince yourself that it does work.

Recall that Alice worked out *c = me (mod n).*

First step is to note that since *c = me (mod n),* then *cd = mde (mod n)* by just putting both sides to the power *d*. How does this help us? Well,

*de ≡* 1 (mod *φ*(*n*)) (that’s how we worked out *d*)

*≡* 1 (mod *(p - 1)(q - 1)*) (that’s just the definition of *φ*(*n*)).

So by the definition of modulus, *de* is some multiple of *(p - 1)(q - 1)*, plus 1. Let’s say that *de = k(p – 1)(q – 1) +* 1. Then obviously *de = (k(p – 1))(q – 1) +* 1 just adding in a couple of brackets, and so the remainder when you divide by (*q* – 1) is still 1, and similarly for dividing by (*p* - 1).

So we’ve shown that *de* ≡ 1 (mod *p* - 1) and *de* ≡ 1 (mod *q* - 1).

Now, there’s a theorem by the 17th century mathematician Pierre de Fermat (you may have heard of Fermat’s Last Theorem? Well this one is called Fermat’s Little Theorem) which states that:

**If *p* is prime and *r* and *s* are positive integers with *r ≡ s (*mod *p –* 1), then we have**

***ar ≡ as (*mod *p*) for any integer *a*.**

So, picking *r* = *de*, *s = 1*, and *a* = *m*, since we have *de* ≡ 1 (mod *p* - 1) we must have

*mde ≡ m (mod p)* (*m* on the right hand side since *ms = m1 = m*)

and similarly

*mde ≡ m (mod q)*

Now, there is a particular case of the Chinese Remainder Theorem to help us here. Note as an aside, that most theorems are named after the person who first stated them, so this name “Chinese Remainder Theorem” seems a little harsh on the first person who described it, Sun Zhu in the 3rd century The reason it is not named after him directly might be the fact that another Chinese mathematician, Qin Jioshao, brought it to more global attention in the 13th century and so it was just attributed to be “Chinese” - but I am doing my bit to give full credit to the inventor by mentioning it here :)

**Anyway, this case of the Chinese Remainder Theorem states that if we have two congruences to modulus *p* and *q* then the same congruence holds to modulus *pq*.**

So, since we have

*mde ≡ m (mod p)*

*mde ≡ m (mod q)*

we have

*mde ≡ m (mod pq)*

So, going right back to the start, *cd = mde (mod n)* and *n = pq,* so,

*cd = mde (mod n)*

*= mde (mod pq)* (since *n = pq*)

*= m (mod pq)* (from the Chinese Remainder Theorem note above)

*= m (mod n)* (since *n = pq*)

and we have shown that indeed, Bob can just work out the message by doing *cd (mod n)* as we suggested. The crucial fact is that he doesn’t need to know Alice’s private key *e*, just remember his own key *d*.

### 6.2.4 Tutorial exercises

Q1. In pairs or small groups, try the algorithm for yourself with your own made-up example, so one of you plays the role of Bob, makes up his key, gives it to the one who plays the role of Alice, who codes her message and sends it back to Bob, who decodes it. Follow the steps precisely and it should work! I suggest, at least for the first time, you choose very small numbers.

Q2. *(Good programming skills needed)*. Earlier you wrote a small program to check for primality. Extend this program so it actually factorises a number and prints the factors on the screen. (*Hint : Take an input n. Find the first factor, say a1 , using the algorithm you have already written. Then repeat your algorithm on (n/a1) and so on).* The output might look like this:

Enter a number : 550

550 = 2 \* 5 \* 5 \*11

Q3. Use the program above to factorise a series of numbers, getting larger and larger each time (so you might try a 2-digit number, then a 3-digit number, and so on). Time roughly how long it takes the computer to work them out (the computer does have an internal clock but you can use a watch if you want), and analyse the time taken to factorise. Concentrate especially on numbers with just two prime factors (find two fairly large primes and multiply them). You should go up to the point where the calculations are just taking too long to bear.

# Section 7 – Cryptography in electronic communications

We’ve looked at quite a few theoretical concepts so far. As we move towards the end of the module, we’re going to change focus a bit and look more at how cryptography is really used and its relevance in our era of electronic communication.

## 7.1 Digital signatures

When you write a letter, you are used to signing it at the bottom. The signature acts as a “guarantee” that it has actually come from you, and that you confirm that the contents of the letter are what you meant to write. How can you replicate this in electronic transmissions? You can’t exactly take a pen and sign an email. Before we look at how it can be done, let’s look at why it’s important to have some way to “sign” our communications.

### 7.1.1 Integrity, authentication and non-repudiation

Theoretically, your signature is something that only you can reproduce, that is it’s not possible to forge it. Although this isn’t strictly true, we’ll assume it’s basically secure in the exposition below, so that you can envisage the signature as being something that only you know, and is impossible to “forge”. Remember at this point, we haven’t touched on how we’ll actually make the “signature”, we’re just discussing why one is needed.

In terms of *integrity*, the recipient wants to be sure that the message has actually come from the person they believe it to be from. In traditional letters, that is what the signature does. Imagine receiving an email from your boss saying that “you are fired”, or a message from your partner saying “goodbye for ever”. If there were no “signatures” in use, how would you know that the email really did come from who you think it has, and it’s not just some hacker trying to upset you?

Or imagine going to a bank and paying money in to your account. The transaction you’ve made is sent electronically by your bank to the central processing centre, so they might send something like (£100, 50117232) where £100 is the amount and 50117232 is your account number. What’s to stop you just sending the same pair to the processing centre yourself to make another £100? Of course, what stops you is that the bank has “signed” their transaction so the processing centre knows the transaction has come from the bank. You can’t replicate the bank’s signature, so when the processing centre gets your message they know you are trying to defraud them because you couldn’t sign it.

A related concept is *authentication*. Recall that this means that you know the message received is the one that was meant to be sent. You know that when you write a cheque, if you change anything, or cross anything out, you have to sign your changes, as well as just the bottom of the cheque. That confirms the changes are what you wanted. Suppose we didn’t sign our changes. Following our examples above, suppose some evil person intercepted a signed message from your boss “I am giving you a pay rise” and changed it to “You are fired”, or from your partner and change “I love you” to “Goodbye for ever”. If there’s no requirement to sign the changes, then you receive the message and believe what it says.

Similarly, in the financial world, suppose you did the same as before, and so the bank sends the message (£100, 50117232) to the processing centre, which they have signed. You intercept the transmission and change £100 to £1,000,000. Although the message is still signed, you have no way to sign the alteration, so again when the processing centre receives it, they know that someone is trying to defraud them.

So, overall, the signature provides an assurance that the message has come from the right person, and hasn’t been altered in any way. So it is genuine.

Another issue we’ll briefly mention here is *non-repudiation* which we also mentioned earlier. Because this message is genuine, then the sender can’t deny they sent it. It’s like buying something online. A vendor gives you a receipt when your money is taken and your goods sent out. This receipt verifies they have taken your money and sent out your goods. So if you never receive them, you have the proof you’ve paid for them and are entitled to them, or a refund. Similarly, you can’t go online, lose a fortune on an online betting site, and then pretend you never paid for it and did the transactions – every time you played, you “signed” that you wanted to make the play, so all your bets were genuinely made by you.

### 7.1.2 Hashing

A common way to create a signature involves the use of a *hash function*. This is a general mathematical concept not just reserved for cryptography, although that is where many of the main uses lie.

The word “hash” just means to “chop and mix”. In our cryptographic sense, it means to take a string of data, and then in some way, chop it up into bits, and mix those bits up, somewhat like we have already considered with some of our cryptographic ciphers.

Hash functions should take in a string of arbitrary length (so any length message) but should always produce an output of fixed length (the “hash length”). (*Aside: You can spot a link here with things we’ve done previously – for example using modular arithmetic to limit the length of the “answer”and then padding it up to the exact length required).*

It is quite common in hashing to use *hexadecimal* to represent the output. What is hexadecimal? It’s used in a huge range of computing applications so it’s worth pointing out what it is. It’s nothing more than representing a number in a different base (just like binary, when we use base 2) – in this case we use base 16 (16 is two bits, commonly used in computing). The word hexadecimal comes from the Greek, hex for 6 and decimal for 10, so 16 overall.

The immediate problem is that we need symbols to represent all the numbers from 0 to 15. It’s no good using “11” to represent 11 – this could be “1” “1”. So, we use letters for the “awkward” numbers. ‘A’ represents 10, 11 is ‘B’, 12 is ‘C’, 13 is ‘D’, 14 is ‘E’, and 15 is ‘F’. The following table summarises the symbols we use, where the top row is “normal” base 10 numbers and the bottom row is hexadecimal.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

So, just like with binary numbers, to convert a “normal” number to hexadecimal we write it as powers of 16. The powers of 16 are 1, 16, 256, 4096, 65536, …. etc. So, for example the hexadecimal string 2E3A8 can be worked out just as in binary: write the powers of 16 on the top, and the string on the bottom.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 65536 | 4096 | 256 | 16 | 1 |
| 2 | E | 3 | A | 8 |

So, we have 2\*65536 + E\*4096 + 3\*256 + A\*16 + 8\*1.

Remembering that A = 10, B = 11, etc, this is:

2\*65536 + 14\*4096 + 3\*256 + 10\*16 + 8\*1

and using your calculator you get 189352. So the hexadecimal number 2E3A8 corresponds to the decimal number 189352.

To reverse the procedure, you do just as we did in binary. To work out what 47512 is in hexadecimal, we first of all find the highest power of 16 that divides into it. 65536 is too big, so we start at 4096.

How many 4096’s divide into 47512? A calculator shows us that it’s 11, with remainder 2456.

Remembering that 11 corresponds to the symbol “B” we have so far the table

|  |  |  |  |
| --- | --- | --- | --- |
| 4096 | 256 | 16 | 1 |
| B |  |  |  |

Now, how many 256’s go into our remainder 2456? Again with our calculator, we work out that it’s 9, with remainder 152. So now we have

|  |  |  |  |
| --- | --- | --- | --- |
| 4096 | 256 | 16 | 1 |
| B | 9 |  |  |

How many 16’s go into 152? We have 9, with remainder 8. So we have

|  |  |  |  |
| --- | --- | --- | --- |
| 4096 | 256 | 16 | 1 |
| B | 9 | 9 |  |

Now how many 1’s go into 8? Well, obviously 8. So we have

|  |  |  |  |
| --- | --- | --- | --- |
| 4096 | 256 | 16 | 1 |
| B | 9 | 9 | 8 |

and hence the hexadecimal corresponding to 47512 is B998.

Again, practice with this and it will soon become second nature. There are some tutorial exercises to try later.

Anyway, we were talking about hash functions, which take a string and convert it into a string of fixed length (often by converting the string to a number written in hexadecimal).

This can be done in a myriad of ways. What sort of functions makes good cryptographic hash functions? Really, they should satisfy these properties:

• The function should “appear” as random as possible, so it is not feasible to compute a message from its hash function

• Small changes in the message should result in totally different hash functions, so an attacker can’t use information from other messages to help them decrypt other messages. So “MEET ME IN THE BAR” and “MEET ME IN THE CAR” should have totally different hash functions, say “19A51DE73208FB21” and “A9B491F110738E2B”. We’ve seen this before too, how one small change in the message changes the output totally.

• Two different messages should not both give the same result when the hash function is applied. If they did, an attacker could switch the original message for his evil message, and it would have the same hash function so would seem to be OK.

It is important to note the distinction between hashing and encrypting. In encryption, we intend to be able to reverse the process later to decode the message. A hash function is one-way – there should be no way of recovering the original message. Hence, passwords are often stored as hashes – when the user enters their password, it is hashed and then this hash is compared with the list of hashes that the network knows. The point is that even if someone finds out the list of hashed passwords, they can’t recover the original password and log in.

Note that in the literature you might also see the word *digest* used instead of “hash” – it’s the same thing, the result of applying the hashing function.

### 7.1.3 Signing and timestamping

There are many ways to digitally sign a document, since it’s really just the creation of an encrypted string and we’ve talked about many ways of encrypting data. We’ll look at the basic idea in the concept of public-key cryptography.

The usual way to sign a message is first of all to apply a hashing function to it, resulting in a hashed message. The sender then encrypts this hashed message, using their own private key, to create a signature. Upon receiving the message, the recipient uses the sender’s public key to decrypt the signature and verifies that it matches the hashed document. This confirms that it did indeed come from the sender.

This seems just the same as with encrypting a whole message and sending it via a public-key method, except that we apply it not to the message, but to the hashed message, to create the signature. Why would we do this? Two basic reasons:

• Hashes are relatively short (remember they are fixed length, so even long messages have a fixed, shorter length) and so it’s efficient to create the signature

• More importantly, we quite often want the message to be readable by anyone, but the signature certifies its authenticity. For example, you have certificates that certify your qualifications. Anyone can read the certificates, and you often want someone to read them (for example, when you apply to a University, you might have to show them to registration staff). It’s the signature that certifies they are genuine.

If you want the message to be secret, then of course you encrypt it in the normal way. But if you want the message to be readable to anyone, but to be certified as genuine, then you just create a signature and send the message as normal text. And of course, you can do both – send a secret message with a signature, so that only the intended recipient can read it, and when they do, they know it came from you because of the signature.

So it’s important to distinguish between the message and the signature. *The signature certifies the message is genuine.*

As well as signing a message, you can also create a *timestamp*. A timestamp certifies precisely when the message was sent, that is just gives a time and date. There’s nothing really to stop a user backdating a message, that is just making up an earlier date on their message. Why would anyone do this? Perhaps to try and claim they did pay their credit card bill on time, not a week late as actually happened.

Timestamping normally works by the hashed message being sent to a trusted third party, a *timestamping authority* (TSA). The TSA creates a timestamp in the usual way from the hashed message, using their private key, just like a signature, and sends it back to the sender. Because the TSA only ever sees the hashed message, they don’t know what the message is, so this is entirely confidential. The timestamp can be recovered by the recipient, using the TSA’s public key, in exactly the usual public-key way. So they have confirmation of when the message was actually sent.

### 7.1.4 Secure Hash Algorithm

The SHA (Secure Hash Algorithm) is a collection of algorithms for hashing. The first, SHA (now usually referred to as SHA-0), was published as a US standard in 1993, intended as a replacement for a previously commonly-used algorithm MD5 (Message Digest algorithm 5). It works on the principles of Rivest, one of the creators of the RSA public-key algorithm. The most common in use now is SHA-1, a slight improvement on the original SHA. It produces a hash of 160 bits and works on any message size up to 264 bits, which is more than enough for any purpose.

SHA-1 is very widely used in modern applications. Copy protection systems (such as those used in the Xbox) often use it, and it is usually used for digital signatures. However, as recently as 2005, certain properties have been shown to be theoretically insecure. Although at this stage, the computing power needed to break it is beyond what is currently available, it has been feared that the techniques there could lead to a deeper analysis producing a feasible way to break it. As such, it is likely that SHA-1 will soon by superseded by an improved version – there are four published versions SHA-224, SHA-256, SHA-384, SHA-512, where the numbers refer to the number of bits in the resulting hash (the chosen sizes reflect the sizes commonly used in the dominant cryptographic algorithms). Collectively these are known as SHA-2. We won’t go into the details of how it works but the next page contains the pseudo-code, published as FIPS-PUB-180-1 by the US Hash Standards Authority. Don’t worry, you aren’t expected to learn this! It’s provided for interest.

*Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating*

*Initialize variables:*

h0 := 0x67452301

h1 := 0xEFCDAB89

h2 := 0x98BADCFE

h3 := 0x10325476

h4 := 0xC3D2E1F0

*Pre-processing:*

append a single "1" bit to message

append "0" bits until message length ≡ 448 ≡ -64 (mod 512)

append length of message (before pre-processing), in *bits* as 64-bit big-endian integer to message

*Process the message in successive 512-bit chunks:*

ak message into 512-bit chunks

bre**for** each chunk

break chunk into sixteen 32-bit big-endian words w(i), 0 ≤ i ≤ 15

*Extend the sixteen 32-bit words into eighty 32-bit words:*

**for** i **from** 16 to 79

w(i) := (w(i-3) **xor** w(i-8) **xor** w(i-14) **xor** w(i-16)) **leftrotate** 1

*Initialize hash value for this chunk:*

a := h0

b := h1

c := h2

d := h3

e := h4

*Main loop:*

**for from** 0 to 79

i **if** 0 ≤ i ≤ 19 **then**

f := (b **and** c) **or** ((**not** b) **and** d)

k := 0x5A827999

**else if** 20 ≤ i ≤ 39

f := b **xor** c **xor** d

k := 0x6ED9EBA1

**else if** 40 ≤ i ≤ 59

f := (b **and** c) **or** (b **and** d) **or** (c **and** d)

k := 0x8F1BBCDC

**else if** 60 ≤

≤ i79 f := b **xor** c **xor** d

k := 0xCA62C1D6

temp := (a **leftrotate** 5) + f + e + k + w(i)

e := d

d := c

c := b **leftrotate** 30

b := a

a := temp

*Add this chunk's hash to result so far:*

h0 := h0 + a

h1 := h1 + b

h2 := h2 + c

h3 := h3 + d

h4 := h4 + e

digest = hash = h0 **append** h1 **append** h2 **append** h3 **append** h4 *(expressed as big-endian)*

Note that in this code, the hexadecimal numbers begin with 0x. This is purely because this is a common standard in many programming languages; the 0 tells the computer that a number is coming, and the x that it is going to be hexadecimal. Also note that the hexadecimal uses lower case letters a-f rather than upper-case A-F. This really doesn’t matter at all, either is perfectly acceptable.

The website http://pajhome.org.uk/crypt/md5/ has an applet that applies SHA-1 to any input string (it also does the previously-used MD4 and MD5)

SHA-1 is commonly used in conjunction with RSA. Refer back to the example we did in the RSA section, and recall that Bob created his own private key *d* and published a number *n.* Let’s suppose Bob wants to send Alice a signed message back. He applies SHA-1 to his message to get the hash *h*. He raises the hash (which is a number, remember, just written in hexadecimal) to the power *d* (mod *n*), so works out

*s = hd (*mod *n)*. This is his signature.

He sends his message to Alice, together with the signature *s* (he may or may not encrypt the message using RSA just as before – if he doesn’t, anyone can read the message, but that might be perfectly OK for the purpose as long as he’s signed it so people know it came from him, just like with your qualification certificates). Alice gets the message with signature *s*, and she works out *se* (mod *n*) where *e* is the public exponent key Bob created. She also applies SHA-1 to the message (decrypting it first if Bob has encrypted it) to get *h*, and if it has really come from Bob, then these two numbers should match *mod n* (I won’t prove this but it follows almost exactly the same lines as with the RSA example we talked about before). If not, then either the message did not come from Bob, or someone has tampered with it.

You can work through an example of this in an extended tutorial exercise following.

### 7.1.5 Tutorial Exercises

Q1) Write the following hexadecimal numbers in “normal” decimal notation.

(i) 2F (ii) 1111 (iii) B9F2

Q2) Convert the following decimal numbers into hexadecimal:

(i) 30 (ii) 256 (iii) 500000

Q3) Convert the binary number 10110101 into hexadecimal (you will probably find it easiest to convert into decimal first, and then convert that into hexadecimal).

Q4) Go to the website given above http://pajhome.org.uk/crypt/md5/ and experiment by typing in strings and using SHA-1. Try two very similar strings (maybe differing by one letter). What do you notice about the outputs?

### 7.1.6 Extended Tutorial Exercise

Work carefully through the following steps to illustrate Bob sending a signed message to Alice. This uses extremely small numbers and a very simple hash function compared to real-life applications!

The scenario is that Alice and Bob both support the same team in a crucial football match. Sadly for Alice, she is stuck at work, while Bob is in front of the TV watching the game. Alice asks Bob to keep her up to date with any goals – sending “HOME” for a home goal and “AWAY” for an away goal. But Alice knows her friends might try to wind her up by sending her false messages pretending to be from Bob. So she asks Bob to sign his messages so she is sure they came from him. That way she can keep up to date with the score.

After a while, the away team score. So Bob wants to send the message “AWAY” to Alice. It’s not a secret message, it doesn’t matter if anyone reads it, so he won’t encrypt the message, but it is important that he signs it so Alice knows it’s genuine.

We’ll use the same keys as before, so Bob has private key *d = 2753* and public keys

*n = 3233* and *e = 17*. Remember, these public keys are freely available, so Alice knows them, but she doesn’t know *d*.

First of all, Bob has to hash his message “AWAY”. The (very very simple, with length only 4) hashing algorithm we will use is as follows – work through each step.

• Convert the message into hexadecimal. Do this by converting each letter into a corresponding 2-digit hexadecimal string, in the following table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I | J | K | L | M |
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 0E | 0F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |

(this is just associating A with 1, B with 2, C with 3, up to Z with 26, but with the numbers written in hexadecimal).

Writing the hexadecimal string for “A” (01) followed by the hexadecimal string for “W”, etc, this should give you an 8 letter hexadecimal string in total for the string “AWAY”.

• Apply the “hash function” which first of all swaps round each pair of symbols in turn, and then swapping over the first half of the message with the second. So, for example, the string “12345678” becomes “21436587” (swapping the first pair of symbols (1,2) and then (3,4) etc) which then becomes “65872143” swapping the halves over. *Ask if this isn’t clear!!!*

We want our fixed hash length to be 4. To achieve this, we will take the modulus mod (65536) (in decimal). This guarantees that, written in hexadecimal, our answer is no more than 4 digits, since every natural number up to (but not including) 65536 can be written in hexadecimal with 4 digits – 65535 is FFFF.

So, follow these steps:

• Convert the hexadecimal string you obtained above into a decimal number

• Work out that number (mod 63335)

• Convert your answer back into hexadecimal.

These numbers are small enough that if your calculator can’t do it, the Windows calculator or Excel can.

You should have a hexadecimal string of 4 characters. This is the hashed message – note: anyone can do this, there is no need for any keys.

Bob will now create his signature.

• Go back to the decimal version of the number you worked out above (i.e. convert it back into decimal). Call this number *h*.

• Calculate *hd* (mod *n)*. You’ll need to do repeated exponentiation to do this, or use a computer program.

• Turn this back into hexadecimal.

This is his signature – no-one else knows *d* so no-one else can forge this signature (unless they purely guessed it – with such small numbers that’s possible, but in real applications it’s too unlikely to consider).

*Note: If you have got it right to this point, this signature should be three symbols long.*

Bob now sends the message “AWAY” together with his signature to Alice. Alice receives the message and is ecstatic, supporting the away team. But wait, is it really from Bob, or a friend playing a trick? She can check.

Firstly, she computes the hash of the message “AWAY” exactly as Bob has done before, so getting the same 4-symbol hexadecimal string as Bob got. So you have *h* just as Bob worked out, there’s nothing for you to really do here!

• Take the decimal version of Bob’s signature, *s*.

• Work out *se* (mod *n)* (remember, *e* is the public key given above)

Does this match the message hash *h (mod n)*? You should find the answer is yes! If not, go back over your calculations. If you try a few times and it’s not working, I’ll help you.

Because it matches Bob’s signature, the message did come from him. Alice can be assured her team is winning and is free to celebrate!

There are a couple of things to note:

(i) This wasn’t that easy and yet the numbers we used were tiny, and the hashing algorithm so simple. Hopefully you are appreciating how impossible it would be to guess or hack keys and signatures with the huge numbers and complex algorithms in use, even for our most powerful computers.

(ii) The hexadecimal-decimal conversion above got a bit confusing and seems almost pointless. However, hexadecimal is much easier for computers to process (remember they are binary devices working on powers of 2, and 16 is a power of 2 but 10 isn’t, so hexadecimal is much easier for them to use than decimal).

## 7.2 Public Key Infrastructure

We’ve made fantastic progress since we started. If you look back on how we started, with the Caesar cipher, progressing through shared key ideas, think about what we’ve got to. We have a system that doesn’t involve any secret key sharing, because every private key never has to be distributed to anyone. Yes, we distribute the public keys, but it’s impossible (or as close to impossible as we can get) to get the private key from the public key, so what does it matter?

This has evolved into a highly secure system. Is it perfect? Sadly not – there is another security issue to consider.

### 7.2.1 Digital certificates

OK, so Bob publishes his public key. But how do we know that when it gets published, this public key really did come from Bob? We haven’t yet set up a way to stop someone pretending to be Bob, publishing their own public key, and hence being able to receive messages sent from people erroneously believing the recipient to be Bob.

The idea is to somehow certify that a public key really did come from the believed sender. How do you certify something? With a certificate.

Suppose you go for a job interview and you claim to have a certain qualification. How do you prove it? Usually, you would be asked to provide the certificate of your qualification. This has to be an original, signed and issued by the relevant authority. The interviewers look at the certificate, and believe it to be genuine since it is issued and signed by an authority they trust. So, they are as sure as they can be that your claimed qualification is true because you’ve got a certificate to prove it from an approved authority.

We can imagine the same idea in our electronic communications. You create your own public key. You then get a *trusted third-party* (i.e. someone else that you trust) to create a file that contains your public key, your details, followed by their (the trusted third-party) digital signature. This is your *digital certificate*.

If someone wants to send a message to you, but are worried if it really was you that issued that public key, they can first of all check with your trusted third-party that the public key really did come from you (so essentially they check your certificate) and then feel confident in sending the message, reassured that it will go to the right person.

The obvious flaw in this is that you choose your own trusted third-party. For an evil hacker, there’s nothing to stop you choosing your partner-in-crime. If they come across as trustworthy, the sender may believe the certificate to be genuine and send the message again (a bit like getting a friend to make up a qualification certificate for you and hoping that the interviewer will trust it and believe it to be genuine). This only works in a *web of trust* where everyone trusts everyone in the network and so everyone is automatically trustworthy. If someone tries to check a certificate and they don’t recognise the third-party as part of the web, then it’s likely to be an attack from outside. Apart from very small networks of colleagues, this isn’t feasible – for example, do you trust every other internet user?

### 7.2.2 Certificate Authorities

A *Certificate Authority* (CA) is an outside authority that is trusted by (almost) everyone in the network. Some businesses might employ their own Certificate Authority for their workforce – an outside client who they know they can trust.

On the web, there are millions of users using different browsers and operating systems. Of course, you want to know that a webpage is genuine and doesn’t contain viruses or other malicious things. So you want to check with a CA it’s genuine. However, if, every time you went to a webpage, you had to be directed to a CA to check a certificate yourself, browsing would become a very tedious business. So most internet browsers (like Internet Explorer or Netscape) have it built in to recognise the CA being used, and automatically check the certificate with that CA, so you don’t have to do anything and your page just opens.

This requires the browser to “trust” the CA. There are many established CA’s that are trusted by the main browsers, such as Verisign. So, when you go to a page certified by Verisign, the browser checks with Verisign the page is OK, and then displays it – you don’t actually notice anything.

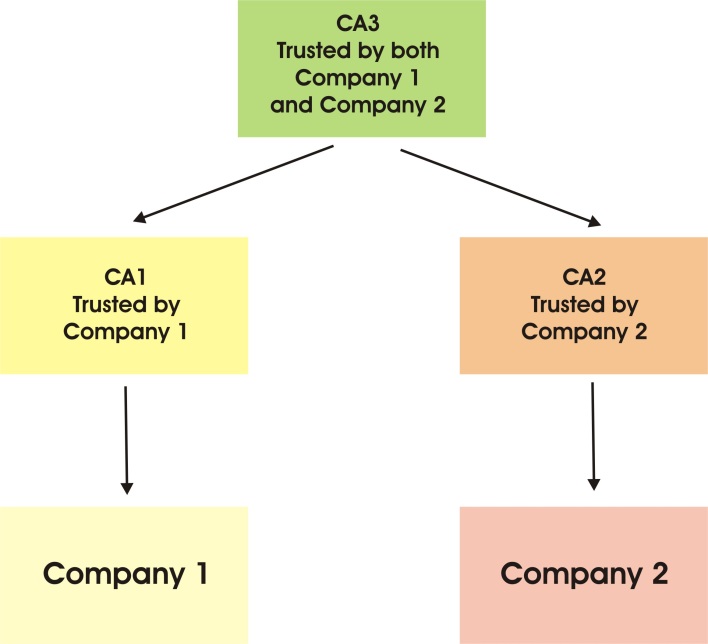
If you go to a page which has been certified by an authority that the browser doesn’t automatically recognise (one that isn’t built in), then it won’t display the page and will ask you if you wish to continue loading. This is when you get those pop-up windows that advise you that “this page may be insecure” or “the certificate on this page could not be verified – do you wish to trust it?” and it’s now over to you to decide if you trust the page enough to open it. In 99% of cases it’s fine to open the page, they just used a CA that wasn’t previously known by your browser, but it’s now your risk.

### 7.2.3 PKI

Suppose we had a situation where two businesses, Company1 and Company 2, employ their own, local CA (the first uses CA1 and the second uses CA2, say). When they try to set up a deal, neither party recognises the other’s CA, they are not one of the well-known ones, and so they are not sure they can trust each other.

But these CAs are companies, and so use public keys to communicate, and so both of these CAs have their own certificates, verified by a larger company, say CA3, which both business trust.

So when the businesses communicate, they may not recognise each other’s CA but they do know that they are verified by CA3, which they trust. So they must be OK. It looks a bit like the diagram on the following page.



In this diagram, the arrows represent that a CA is certifying the company below them. So CA1 certifies Company 1, and CA2 certifies Company2, and CA3 certifies CA1 and CA2. When Company1 and Company2 try to communicate, they don’t recognise the certifiers. But then they check higher up, and they discover that CA3 who they both trust, says that CA1 and CA2 are OK. So they are both happy to communicate.

You can hopefully imagine extending this further. Someone even higher up verifies CA3, and so on. Then imagine millions of companies communicating. There is a huge mess of arrows, starting from the thousands of small CAs, building up to the bigger and bigger CAs that are trusted by more and more people. Ultimately there might be one “super-CA” that is trusted by everyone. If everyone on the network is using Windows, then this might come down to Microsoft itself (as scary a thought as that might be). The idea, is that if Microsoft (through the CAs it has verified, who have then verified more, and so on) can eventually verify your small CA then it must be OK. If you encounter a CA that isn’t anywhere in this huge chain, then it’s unlikely to be trustworthy, since the “super-CA” doesn’t even trust it, so why should you?

This whole network (mesh) of CAs and how they are connected is called a *public-key infrastructure*. The pattern of arrows describes how the CAs are connected, who has certified who, and so on.

It’s a bit like you presenting a certificate. The interviewer isn’t sure about the authority that gave you the certificate, if they are really genuine. So he goes to a higher level and asks their regulators if this authority is genuine. If they say yes, and he trusts them, then everything’s OK – the authority is genuine, and so their certificate is genuine. If he still is short of trust, he takes it higher and higher if he needs to, until he decides that someone he does trust certifies everything in the chain is OK.

As another example, suppose you ask me for a reference but you aren’t sure you can trust me to write something fair. So you go to my boss (Head of Department) and he tells you that he knows I will be fair. But you don’t even trust him. So you ask his boss (the Dean) if he trusts the Head of Department. You do trust the Dean, and he says yes. So, you are happy that the Dean trusts the Head of Department. And the Head of Department trusts me. So the Dean trusts me. You’re now satisfied you can trust me and I write my reference happily for you.

Think about this for a while (and we’ll discuss it in lectures) and it will soon become obvious!

### 7.2.4 Tutorial exercise

Find around 10 examples of Certificate Authorities (VeriSign is probably the most famous one, to get you started). For each one, write one or two sentences saying something about them (when they were formed, number of users, any interesting fact).

## 7.3 Electronic cash

“Money makes the world go round”.

### 7.3.1 Cash

For thousands of years, the concept of wealth and money has been pivotal to mankind. Rightly or wrongly, money dominates many people’s lives. For most of history, money has existed purely in a physical form. Notes and coins had a certain value and you used them to buy the things you want. Your wages would be given to you in notes and coins, and you could then spend that as you wished.

In the 20th century money started to take a slightly different form. Not everything was now “cold hard cash”. Cheque books were invented – you could write a cheque for a certain amount and give it to someone, which authorised your bank to transfer that amount amount of money to your payee. Wages started to be paid directly into the employee’s bank account.

Then debit and credit cards came along – now you could just use this card to automatically pay for things, there was no particular need to carry notes and coins with you, when you wanted to buy something you could use your card. When the card transaction went through, it was passed to your bank who took the money from your account and transferred it to the seller’s account.

Also if you wanted to get some hard cash, you can just use this card 24 hours a day in any ATM machine to get the cash you wanted. No need to queue at the bank to withdraw your money, just use the machine.

All in all, the actual coins and notes were being used less and less. Then along came the internet….

### 7.3.2 Electronic cash

The majority of financial transactions nowadays are conducted electronically. People use online banking, pay their bills through the internet, buy things online, even day-to-day things like buying your groceries can be done online. The actual changing hands of physical cash, although still important, is not the predominant force in the economy any longer.

Electronic transactions really just use numbers. If you have £110.53 in your account, your bank account isn’t actually an envelope with £110.53 in it. It’s really just a number to the bank to show your worth, part of the billions of pounds they actually have. If you buy something online for £20, they don’t actually take a £20 note and remove it from your account, they just adjust the number associated with your account to be £20 less, so £90.53 in our example. Similarly the bank of the seller adjusts their balance by £20 upwards.

All this means that these transactions have to be conducted by sending account numbers and balances and personal details electronically. Naturally, security is of absolute paramount importance. Of course you don’t want your credit card details sent by plain text, you want them encrypted so securely that even if someone does intercept them, they have no hope of extracting them.

When sending these details, to pay by something by credit card, say, your card has to be authorised by your bank. First of all, it must be a valid card (so not expired, for example) and you must have sufficient funds in your account to make the transaction. It also checks for fraud – you have to provide the address associated with the card, and also prove you have the card in your possession and haven’t just obtained the card number from somewhere (usually by typing in the expiry date and the Card Verification Value (CVV) which is usually the last three digits on the reverse).

The actual architecture of the payment and verification structure varies between companies, but typically a business will purchase a product from a company such as WorldPay, which takes care of all the processing, leaving the seller free to concentrate on their products, shipping, etc. This intermediary software receives the buyer’s credit card details, checks with their bank that it is OK, and notifies the banks of the buyer and seller to adjust their balances accordingly.

It is obvious that this procedure has to be made as secure as possible. The potential for fraud if someone intercepts your credit card details is immense. Hence, these payment systems must use the most cutting-edge cryptographic techniques to ensure that the system is completely secure.

### 7.3.3 Tutorial exercise

In small groups if you like, list some examples of when you have used electronic communication for financial transactions. Don’t just limit to buying something over the internet – do you use online banking? Or use a debit/credit card in the shops, or at an ATM to withdraw money?

## 7.4 Quantum cryptography

This brief section is really just an aside and is provided for motivation and interest only.

### 7.4.1 History and background

The security of the algorithms we have talked about recently, such as RSA, is based on mathematical properties that are easy to formulate but practically impossible to solve (such as factorising a very large number). Quantum cryptography is an entirely different approach to cryptography and is based on physics and quantum mechanics. It was first thought of in the 1970s by Stephen Wiesner at Columbia University, although his work was not published until the 1980s. Based on this work, Charles Bennett (IBM) and Gilles Brassard (Montreal) published the first real quantum cryptographic scheme. Artur Ekert (Oxford) developed a slightly different approach to quantum cryptography in 1990, whilst working as a PhD student.

The technique is still very much in development. The transmission of the data has been made able to work over a distance of a few hundred miles, using fibre optic cables – thoughts of using satellite technology are being considered.

### 7.4.2 Photons and Uncertainty

A photon is a tiny particle of light. As an atomic particle, it has certain properties, such as its polarisation (is it up or down, left or right?) The idea is to use photons to transmit data rather than computers. Rather than sending a stream of 0s and 1s, we send a beam of light as a sequence of photons, where each photon can be associated wth a binary number (e.g. if its polarisation is “up” it assumes the value 0 and if its polarisation is “down” it assumes the value 1).

Hence, to transmit a message, key, or anything you wish to send, you could send it via a fibre optic cable as a sequence of photons.

The really big advantage of doing this comes via quantum mechanical properties. Heisenberg’s Uncertainty Principle states that even the act of observing such photons affects them and their polarity, say (this is not a Quantum Mechanics module and so I will not go into any further detail). So if a malicious hacker even tries to look at the communication, he will destroy its configuration and the fact that someone has tried to intercept or eavesdrop will be immediately obvious to the receiver.

This is all I want to discuss with regard to this subject, for the purposes of this module. This is a developing area, still very much in its infancy, and to get deeply involved would turn it into a physics module. Please do though, feel free to do some research and find out anything you can!

# Section 8 - Cryptographic protocols and systems

We’ll now move on to look at a few current cryptographic protocols and systems and give you the idea of how they work.

## 8.1 Protocols and systems

First of all, we should just clarify the use of the terminology “protocol” and “system”.

### 8.1.1 Protocols

The word “protocol” has several meanings in English, but in terms of our area of interest, a protocol can be considered as a set of rules defining the communications happening in our network. The protocol should describe how the communication is to be carried out, explaining the encryption and decryption methods, what to do in case of error, how the keys are going to be used, and exactly how the message is to be transmitted.

So basically, a protocol is a description of how a communication works. We’ve already informally defined several protocols in our discussion so far. For example, we explained the RSA/SHA protocol. We described how each party communicates, how they convert their messages and how they send them.

Remember – a protocol describes, in formal terms, the communication process.

### 8.1.2 Systems

A cryptographic system refers to every aspect of the network and the cryptography being used. It does not just restrict itself to the communication, but also the storage of data, and how the concepts of authorisation, authentication, non-repudiation etc are all implemented, so it can be considered as a description of the entire system that we are using.

In actual fact, these words tend to be variously misused, overused, underused, confused, depending on the author and the process, and actually are often used in place of each other. Don’t worry about the technicalities, just understand that a protocol should basically define the set of rules for communication, and a system is the overall system as a whole.

## 8.2 Some cryptographic protocols and systems

Let’s look at some protocols and systems that are in use today and compare the way they go about security and communication. We’ve already studied public-key RSA/SHA as our main focus – what else is there? We can’t cover everything but let’s look at a few examples. We won’t go into the full details, but will overview the approaches and ideas.

### 8.2.1 Kerberos

Kerberos is an authentication protocol, which means it is used for two parties communicating on a possibly insecure network, to establish each other’s identity. It was originally developed in the 1980s at the Massachussetts Institute of Technology (MIT). The name Kerberos is derived from Greek mythology, as are many names in computing – Kerberos (also spelt Cerberus) was in legend a three-headed dog who guarded the entrance to Hell. As an unusual fact, the Kerberos system was banned from being exported from the US in the early 90s and so was only available for use there. This was because, as a security system, it technically fell under the definition of “munitions” (weapons) according to US export laws, because of its potential to be used as a security device for planning of attacks.

Kerberos itself acts as the trusted third party in all the communications. Every user on the system has their own secret key – their secret password that they chose and use to log in to the network – and their computer (client) creates its own secret key from the password, generally achieved by applying some hash function to it. Remember that passwords on a system are never stored as plain text, just the hashes are stored, from which the original password cannot be extracted.

The basic principle behind Kerberos is that, once a user’s identity has been confirmed, it issues a “ticket” to that user allowing them to use the communication facilities provided. This ticket is valid only for a short period – this helps to prevent a hacker analysing a transfer and coming back some hours later having broken it – the ticket is no longer valid and so they cannot disrupt your communications.

The Kerberos server itself is split into three basic sections – initial authorisation, ticket issuing, and overall communications server. Each procedure follows similarly – the user uses their secret key to establish their authority with the next stage, just as we have talked about before.

First of all, Kerberos establishes that the client (the computer the user is using) is a properly registered computer on the network, so it’s not some hacker from outside. It uses DES or some equivalent standard to achieve this using the secret keys. Next, the server issues the client with a ticket, to which the client responds (using their secret key) to confirm authentication. They also timestamp the ticket and establish how long it is valid for. Finally, having done all of this, there is enough information for the client to confirm their authority with the central server, using their ticket, and they are now able to begin communications. For this, they are issued with a session key, which allows them to communicate, using the Kerberos main server, for the duration of the session.

As soon as the ticket runs out, you have to get a new one (a bit like travelling on a 24-hour railcard – when the time is up, you have to buy a new one) so need to be authenticated again. This has the advantages we discussed in the previous paragraphs – if a hacker manages to break into your communication, it’s probably too late and your ticket has expired.

For most of its development, Kerberos has run on a secret (private) key basis – which does mean the sharing of private keys somehow. Recent developments have included public key methods to improve the security of the system.

There are a couple of immediate drawbacks with Kerberos. One is the total reliance on the central Kerberos server itself. If this was to go down (and servers do go down) then no-one can communicate at all, since all communications run through the server. If you are running this on a vital network, it’s a bit like the whole internet going down and nothing being available. This can be alleviated to some extent by employing multiple servers, but of course this has cost implications for the network using it.

The second concern is the timestamping. Tickets are valid for a specified period of time. This does require all the computers in the network to be synchronised with their clocks. You might (some of us do….) have family in southern Siberia which is 8 hours ahead of GMT. Let’s say we have a special ticket that allows us to contact Siberia for free in December. Late on New Year’s Eve I use my ticket to contact the family. But it’s already January according to their time and ther computer (remember they are 8 hours ahead) and so to them the ticket has expired, even though it seems OK for me.

So it is important to establish a synchronised time system between everyone in the network and make sure everyone is working from the same time frame (say GMT).

Kerberos is still being developed all the time. It provides a secure system for people to communicate and be sure of others’ identities, so that users are reassured that who they are talking to really is the person they want to talk to.

### 8.2.2 Pretty Good Privacy

Pretty Good Privacy (PGP) was developed by software designer Mike Zimmerman in 1991 and has been constantly developed since - it is now the mostly widely-used email encryption system in the world. The name “Pretty Good” was apparently inspired by a fictional shop “Ralph’s Pretty Good Grocery”. Zimmerman found himself embroiled in a court battle with the US authorities after publishing his software - as with Kerberos, the Government legislation was that it constituted a “munition” and so technically, by publishing it worldwide, Zimmerman was accused of essentially exporting weapons. The case was eventually dropped in 1996.

PGP can be used to encrypt any data, but by far its most common use worldwide is in email applications. The majority of the main email programs (Outlook, Eudora, Mutt etc) use a plugin that utilises PGP to ensure the mails are sent securely. Its operation uses many of the cryptographic techniques we have already considered, so is a nice example to illustrate the concepts in this module coming together.

It uses a symmetric (private-key) cipher called IDEA (International Data Encryption Algorithm) for the encryption and decryption of the actual message. We won’t discuss the details of IDEA here – it is a widely-used block cipher. This cipher replaced Zimmerman’s original cipher “Bass-o-matic” which he invented himself but was soon found to have serious flaws.

Further to the encryption of the message, each message is signed by the user, using some hashing algorithm, and the signature is transmitted along with the encrypted message, using RSA in exactly the way we have described. This use of a public-key algorithm does of course lead us to needing certificates, as we have discussed. Most versions of PGP take a web-of-trust approach where the certificate can be issued by another user, but more recent implementations have started to use Certificate Authorities and are moving towards a more typical Public Key Infrastructure, again exactly as we discussed before.

You might ask why the combination of symmetric-key and public-key algorithms? For one, using two different approaches and methods in the same system is obviously harder to completely break. Secondly, symmetric algorithms are generally much faster than public-key algorithms, although they often have more security worries – with the amount of email being sent and the instant transmission required, speed is an important factor.

### 8.2.3 Secure Socket Layer

Secure Socket Layer (SSL) was developed by Netscape in the mid-1990s and is now used worldwide as a communications protocol, particularly used for e-commerce and financial transactions. All of the major financial institutions (Visa, MasterCard, etc) have approved it for use in e-commerce. You might also see the abbreviation TLS (Transport Layer Security) – this is a development on the protocol SSL 3.0, but the overall concept is still usually referred to as SSL.

Webpages utilising SSL for their communications are often described as “secure sites” and you see the webpage change from *http://* to *https://* where the *s* stands for “secure”. You often see this on financial sites, or when you pay for your product you might get the symbol “you are now being transferred to our secure server”. You might also have seen the pop-up box “this page contains both secure and non-secure items. Do you wish to continue?”. This means that aspects of the page use SSL, but not all, this is generally considered poor design.

Essentially, SSL provides the basis for the communications to be carried out. At the start of an intended communication, it sets up the appropriate method of communication to be used. The current versions support public-key cryptography like RSA, symmetric-key cryptography like AES, and hash functions like SHA. Remember, that each method has its own advantages and disadvantages. The SSL server “negotiates” with the user’s computer to establish which system they will use, and then carries out the communication using the agreed method, choosing the appropriate keys and carrying out all authentication etc required.

The user’s computer (client) sends a “hello” message to the server, listing which ciphers it knows, and various technical data. The server responds with a message choosing the methods that will be used, according to the options provided by the client. They then use certificates to authenticate each other, and then establish a secret key to be used, depending on which cryptographic method is going to be used. The user is then authenticated and ready to start sending messages through the server to other users.

A “record” of all communications made is stored. Each record can be encrypted and a specifically generated code attached, for extra security.

An open-source (i.e. free!) version of SSL has been developed, called OpenSSL, predominantly for use in Unix computers but also in Windows, which allows anyone to create their own SSL-protected communications.

### 8.2.4 Internet security

Everyone knows that the internet is not a completely safe place. Hacking happens, viruses spread, there are scams and cons, and criminal activity does happen. Although, of course, these happen out in the physical world as well: in fact to a far deeper extent. Most communications over the internet are secure. It is important to realise that the level of security, and the methods used, depend a lot on what you actually want to transmit. For some communications, (e.g. instant messaging among friends) security isn’t that important but speed is. For others, such as finanical transactions, you would be happy to wait 10 seconds for a secure transmission rather than an immediate, unsafe transaction.

Which protocol you choose to use depends on what you actually want to achieve. There is no “best” cryptographic protocol.

### 8.2.5 Tutorial exercise

Find out something more about Kerberos, PGP and SSL. How widely-used are they? What are their main features? What drawbacks do they have? A few lines on each is fine.

To get you started, you can look at the webpages:

* http://web.mit.edu/kerberos/
* http://www.pgp.com/
* http://www.ssl.com/

Also see if you can find web pages that are critical of the systems and mention problems.

# Section 9 – Cryptanalysis

Throughout the module we have touched upon various methods of cryptanalysis – how might you go about trying to decode a secret message? How can you attack a communication? Much of this section will be summary material as we pull the whole idea of cryptanalysis together.

## 9.1 Statistical, historical and brute-force attacks

First of all, let’s briefly survey some of the more “obvious” ways to try and decrypt a message we’ve managed to intercept.

### 9.1.1 Statistical analysis

With our basic ciphers, one important way to attack them is to try and use some sort of statistical analysis to help us guess “likely” keys or encryption methods. The way to do this is to search through the encrypted message looking for patterns. In our very simple Caesar cipher, for example, we noted that ‘E’ is the most common letter of the alphabet and so the letter that appears the most in our encrypted string is quite likely to correspond to ‘E’. If not, it probably corresponds to another very common letter such as ‘A’ or ‘S’.

Similarly with Kasiski examination, we examined the encrypted message looking for patterns which might correspond to repeated words. This helped us guess the length of the keyword which in turn gives us more information to help us decode the whole message.

In the modern age, statistical analyses are less likely to be useful as the algorithms have developed. But they can still be powerful. In trying to break into a code or message, you might be prepared to take months solving it (imagine a war situation, say). If you can just get any sort of inkling, a clue, even an idea of an approach, then you’re on the way. It might not work for the first 1000 goes, but then it might. Remember, in cracking a message, there are many people working on many different aspects. Any progress or lead, or possible hint as to how to progress, however small, can advance the cracking procedure – these small steps can eventually lead to the final solution.

### 9.1.2 Brute-force attacks

A brute-force attack on a system is simply trying every single possibility until you eventually encrypt the message. So for example, when the communicators share a key, you try every possible key on the messages they send until you get something where they are all meaningful.

If the key was of length 3, using the English alphabet, then there are only 26 \* 26 \* 26 = 17576 possible keys to check. This is trivial for a computer to run through them all, produce the output, and for us to see if any output makes sense. But of course, current systems use keys of huge length, and the number of possibilities to check is simply beyond what we can do with current computing power. Yes, as computing power increases, we may need to increase key lengths, but for now they are OK.

Note that this approach only works when the same key is used more than once. There are many possible decryptions of a string – it might be we guess totally the wrong key but by fluke one of the messages decrypts to an English sentence. But we should know this can’t be right as none of the other messages make sense with this key. The aim is to find the key that decrypts *all* the messages. With the one-time pad, this brute-force approach doesn’t work. Every combination of letters is equally likely for each message – and we change the key every single time. So for a message of length 50, any single string of length 50 is a possible decryption, and you can’t gain any information from other messages because you change the key each time.

For all algorithms in common use, they are all aware of brute-force attacks and the keys are designed to be long enough that a brute-force attack is quite simply impossible, with the power available. An algorithm is said to be secure if there is no better way known to crack the method, than simply by brute-force. So, if this is the best known way to break it, and that’s infeasible, then it can be considered safe.

If an attack is found that takes less time than brute-force in any way, then the cipher is considered to have security weaknesses.

### 9.1.3 Mathematical attacks

Remember that a lot of the algorithms in use rely on mathematical problems that are easy to formulate but infeasible to solve, at least with the algorithms that we know and our current computing power. RSA is the perfect example – it relies on the problem of factorising a huge number to ensure its security. Although the sender can easily compute this huge number, no-one else is able to factorise it, the way to do it just isn’t available to us.

There are many other algorithms which work on a similar idea – a mathematical problem that is “impossible” for us to solve, such as computing a logarithm or many problems in elliptic curves.

We could use brute-force to try and attack these – such as trying every possible divider to try and factorise a number. But the numbers are chosen so big that this isn’t really feasible and we are unlikely to get to the solution. There are algorithms that help make the task a bit easier, but so far nothing has been discovered that allows the factorisation to be done in an “easy” way, or in feasible time.

But this leads to another method of attack – mathematical research. If you could find a way to factorise large numbers quickly, you could break some of the most important, powerful systems in the world. That is why intelligence organisations such as GCHQ (the British Intelligence Centre, based in Cheltenham) employ mathematical researchers to work on such problems.

So far no such algorithm or method has been found, and no-one knows if one could even theoretically exist. But maybe one day it will be found. So cryptanalysts don’t just work on decrypting a particular transmission, they are also looking at general techniques that could be developed to be used in the future.

### 9.1.4 Physical attacks

This is a maths and computing module so naturally we have focussed on the theory and the computational methods behind possible attacks. But let’s not forget this is all being conducted in our world, it’s not a fantasy tale.

You may have heard stories about MI5 spies leaving briefcases with sensitive data in a railway station or bar by mistake. If you want to obtain a key and you know who has it, you could find them and hold them at gunpoint until they reveal it. By the way, I do not advocate you do this!!!!!

But the point is that these physical matters do have to be taken into account as well – there’s no point taking every possible security measure with regard to the electronic communication if your agent then walks around with a piece of paper detailing his secret key and all the security details. So the physical measures needed to keep keys secret do all have to be addressed.

Remember – these communications are not separate from life, they are part of it.

## 9.2 Passive and active attacks

The essential role of the malicious hacker is to intercept the transmission in some way and learn or gain something from it. There are really two fundamental ways they can do this – to gain information from the transmission they have intercepted, or to alter it and cause chaos.

### 9.2.1 Passive attacks

The first type of “attack” to infiltrate a cryptographic system is to “listen in” to the communication and try to work out the protocol and keys etc using the communications that you hear. So, as the messages are being sent, you read them, but don’t in any way attempt to alter them, stop the transmission, or to join the network in any way yourself. You simply read them and try to decrypt them so you can work out what’s going on – if you manage to decrypt them and work out all the keys etc then you can keep listening in and read every message being sent.

This practice is called “eavesdropping” (this is a normal English word, not just for cryptography, meaning to listen in to others conversations). Following the Alice and Bob convention, an eavesdropper is usually referred to as “Eve” (short for eavesdropper!).

Such attacks, where one doesn’t attempt to influence the communications or the network, but just listens in and reads messages, are called *passive attacks*. Telephone lines are suspect to passive attacks – if someone can listen in to your phone calls then they can gain vital information from what you say, and you don’t know anything about their eavesdropping. A recent example to illustrate that this goes on was the Italian match-fixing trial. The chief of one of Italy’s top football teams, Juventus, was having his phone calls listened to by investigators, ironically investigating a totally separate issue. To their surprise, the investigators heard the chief discussing the appointments of referees and influencing the Italian Football Association to appoint “favourable” referees for their games. As a consequence of the eavesdropping of these telephone calls, Juventus were stripped of their Championship and relegated to the second division.

Passive attacks are very unlikely in modern cryptographic systems because the level of encryption is so deep that encryption is virtually impossible. In RSA, for example, the message is encrypted using keys that are computationally infeasible to find. So, current systems are considered secure against passive attacks.

### 9.2.2 Active attacks

As we said above, cryptographic systems now are secure against pasive attacks. Listening in to, and reading, messages doesn’t help – they are encrypted so powerfully that decrypting them is practically impossible, so you can’t gain anything by being an eavesdropper. More sophisticated styles of attack are needed. Again, we’ve really discussed all of this in our previous work, so this is just a quick overview.

In contrast to a passive attack which just listens in, an *active attack* is one in which the malicious hacker actually alters the communications, or enters the network, so is now playing an active role, not just listening. An active attacker, again following the Alice and Bob notation, is usually referred to as Mallory – short for *malicious attacker*.

As an example, in a simple system, Mallory might have intercepted Alice’s password sent in a previous communication. He then gets on to the network, pretends to be Alice, and sends a message to Bob. He’s unsure if it’s really Alice, so he asks “her” to enter the password in a reply message. This Mallory happily does – Bob thinks Mallory is Alice. We discussed digital certificates and the public-key infrastructure of Certificate Authorities as a way to prevent this. Or he could intercept a transmission, modify it in some way, and then send it on. We talked about the use of digital signatures to ensure this can’t happen – if the message is changed in any way, then the signature won’t match and Bob knows the message has been tampered with.

The fact is, that as our systems develop, and our understanding of cryptography grows, so does that of the malicious hackers. As their skills grow, there is always the possibility of them coming up with an attack that we hadn’t considered. That is why cryptanalysts research this – they test the systems, try to hack them, try to come up with new methods of attack – if the researchers on your side can do it, then so can a malicious hacker on the outside. Active attacks are substantially harder to protect a system against those passive attacks.

### 9.2.3 Tutorial exercise

As we approach the end of this module and you are busy with your assignment, there is no specific technical exercise for you to do, so breathe easy!

But, I’d like you to think in general about attacks. Think about some of the attacks we have proposed during this module, and some we discussed. For example, how was Enigma broken? By a passive or active attack? Think about the approaches we have taken to make a secure system, signatures and certificates and so on. Can you think of any other ways to go about trying to hack a system?

Some hackers do it to gain information or cause damage. Others do it out of curiosity to “see if they can break it” or for self-pride in their computing skills. Can you find any examples of where systems have been broken? What was the intention of the hacker – to cause damage, or out of curiosity to see if they could hack it?

Write down any thoughts you have or anything you find out.

# Section 10 – Conclusion

The aim of this module has been to give you an understanding of what cryptography and protecting our data is all about. You don’t really notice all these things when you send an email, for example. You just think that it has been sent and your intended recipient received it. But there is a huge amount of encryption, security checks, in the whole process. Why do you not notice it or think about it? Because the security is so strong that you just assume it will work fine.

Cryptography is a rapidly-developing subject. The almost unbelievable growth in computing and the internet means that to keep pace, the subject has to grow incredibly as well, which is why vast amounts of money and time is spent on research and developing our security systems.

What does the future hold as computing power continues to grow? Will we find solutions to the mathematical problems that currently protect our systems? Will quantum cryptography be made to really work and take over? These questions are waiting to be answered.

In summary, by taking this module you have studied one of the most dynamic, developing, important subjects in computing and indeed the world!