# Computer Security

(complete)

**Andrew D. Ker**

16 Lectures, Michaelmas Term 2014

Department of Computer Science, Oxford University

# Contents


**1 Introduction** **1**

1.1 Preliminary Material . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.3 Aspects of Computer Security . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.3.1 Authorization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3.2 Confidentiality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3.3 Integrity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3.4 Authentication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3.5 Non-repudiation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.3.6 Availability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.3.7 Other Aspects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Access Control** **9**

2.1 Modelling Access Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Modelling Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3 Enforcing Access Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.4 Case Study: UNIX Permissions . . . . . . . . . . . . . . . . . . . . . . . . . 16

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**3 Symmetric Key Ciphers** **19**
3.1 Formal Definition . . . 20
3.2 Attacks . . . 20
3.3 Weak Ciphers . . . 23
3.4 Perfect Security . . . 25
3.4.1 Shannon Security . . . 26
3.4.2 The Vernam Cipher and the One-Time Pad . . . 28
3.5 Practical Security . . . 29
3.5.1 Feistel Structures . . . 30
3.5.2 DES . . . 32
3.5.3 AES . . . 36
3.6 Repeated Encryption . . . 36
3.7 Encrypting Longer Messages . . . 38
3.8 Stream Ciphers . . . 40
3.9 Concluding Remarks . . . 42
Bibliography . . . 43

**4 Cryptographic Hashes** **45**
4.1 One-Way Functions and Cryptographic Hashes . . . 46
4.1.1 Proofs About Hash Function Properties . . . 48
4.1.2 From Compression Functions to Hashes . . . 49
4.2 Exhaustion Attacks on Hashes . . . 50
4.2.1 The Birthday Paradox . . . 51
4.2.2 Consequences of the Birthday Paradox . . . 53
4.3 Hashes and Passwords . . . 53
4.3.1 Offline Attacks . . . 55
4.3.2 Password Eavesdroppers and Lamport's Scheme . . . 57
4.3.3 Authentication Tokens and Biometrics . . . 58
4.3.4 The LM Hash . . . 59
4.4 Hashes and Key Generation . . . 60

4.5 Hashes and Message Integrity . . . 61
4.5.1 Do Collisions Matter? . . . 63
4.6 Construction of Cryptographic Hash Functions . . . 63
4.6.1 The Merkle-Damgård Construction . . . 64
4.6.2 The MD4 Hash . . . 65
4.6.3 Successors to MD4 . . . 66
Bibliography . . . 68

**5 Asymmetric Key Ciphers 71**
5.1 Number Theory Primer . . . 72
5.2 The RSA Cryptosystem . . . 73
5.2.1 Correctness . . . 75
5.2.2 Efficiency . . . 75
5.2.3 Security . . . 78
5.3 RSA Signatures . . . 80
5.4 Attacks on RSA . . . 80
5.5 Other Asymmetric Ciphers . . . 83
5.6 Key Exchange . . . 84
5.6.1 Security . . . 85
Bibliography . . . 85

**6 Message Authentication & Digital Signatures 87**
6.1 The Dolev-Yao Model . . . 87
6.2 Authenticity and Integrity: MACs . . . 89
6.2.1 MACs Based on Ciphers . . . 91
6.2.2 MACs Based on Hashes . . . 92
6.2.3 Other Types of MAC . . . 93
6.3 Authenticity and Non-Repudiation: Digital Signatures . . . 94
6.4 RSA Signatures . . . 96
6.5 Signing and Encryption . . . 99
6.6 Alternative Signature Schemes . . . 102
6.7 Concluding Remarks . . . 102
Bibliography . . . 102

**7 Security Protocols** **105**

7.1 Entity-Entity Authentication . . . 107

7.1.1 Unilateral Authentication . . . 107

7.1.2 Bilateral Authentication . . . 113

7.1.3 On Nonces . . . 116

7.2 Key Distribution . . . 117

7.3 Mediated Authentication . . . 120

7.4 The SSL/TLS Protocol . . . 122

7.5 Recent Attacks on SSL/TLS . . . 125

7.5.1 Attacks on Certification Authorities . . . 126

7.5.2 Attacks on SSL/TLS Block Modes . . . 127

7.5.3 Attacks on SSL/TLS Compression . . . 128

7.5.4 Attacks on RC4 . . . 128

7.5.5 Denial-of-Service Attacks . . . 129

7.6 Concluding Remarks . . . 129

Bibliography . . . 132

**Index** **133**

# Chapter 1

# Introduction

## 1.1 Preliminary Material

**Computer Security** is a part B third year option for undergraduates in Computer Science or Mathematics & Computer Science, and a Schedule B option for the taught Masters in Computer Science. There are 16 lectures (three per week in the first two weeks, then a planned two per week for weeks 3–7). There are four classes (weeks 3, 4, 6, & 8) and two practicals (weeks 2–4 and 6–8).

### Prerequisites

Most chapters involve a little discrete probability, including conditional probability and random variables. There is nothing more advanced than would be covered by any introductory probability course.

Chapter 5 involves some modular arithmetic. Undergraduates will have seen enough in the first year discrete mathematics (CS) or algebra (M&CS) courses. These lecture notes include basic definitions but students who have never seen modular arithmetic, or who have forgotten it, may wish to do some background reading.

The practical work is in Java. No advanced programming techniques will be used, it should be sufficient to know the basic syntax.

## Syllabus

Aspects of security, security models. Tools for achieving particular security goals, attacks, and countermeasures: one-way functions, symmetric and asymmetric block ciphers including key generation and block modes, keyed hashes, digital signatures, simple key exchange protocols. Applications in practice.

## Synopsis of Lectures

**Introduction.** Motivating example, definitions of security aspects.

**Access control.** Models of security and the example of Unix permissions.

**Symmetric key ciphers.** Block ciphers, Kerckhoffs' Principle, attack models, examples of attacks (including meet-in-the-middle). Perfect security, Shannon's conditions, Vernam cipher and the one-time pad. Feistel networks, DES, state-of-art encryption. Block modes, stream ciphers.

**Cryptographic hash functions.** One-way and cryptographic hash functions, relationships with other security properties; attacks on iterative algorithms. Hashes for password storage and key generation: offline attacks, strengthening by salting and stretching, examples in practice. Hashes for message integrity, collision resistance. The Merkle-Dåmgard construction, padding, MD4 and its successors.

**Asymmetric key ciphers.** The RSA cryptosystem: proof of correctness, discussion of efficiency. The RSA security assumption, relationship with integer factorization. Discussion of appropriate key size. Homomorphic property, number theoretic attacks, PKCS v1.5 padding. Brief survey of alternative public key ciphers. Brief description of DHKE.

**Message authentication & digital signatures.** The Dolev-Yao model. Message authentication codes, HMAC. Digital signatures, attack models. RSA signatures, textbook weaknesses, and PKCS v1.5 padding. Combining signatures and encryption. Brief survey of alternative signature schemes.

**Protocols.** Entity-entity authentication protocols given shared secret or public keys, weaknesses and attacks. Key distribution and certification, chains of trust, PKI. Mediated authentication protocols, weaknesses and attacks. The SSL/TLS application layer protocol.

### Reading Material

The main course text is

- "Network Security: Private Communication in a Public World"(Kaufman et al., 2002), second edition.

It is reasonably cheap (from £30) and very readable. It covers nearly everything in the course, sometimes in good detail and sometimes only in outline, except for the lecture on access control which is barely covered at all. The authors seem reluctant to use equations and the book sometimes lacks formality, but these lecture notes are intended to bridge that gap. Published in 2002 but not noticeably out of date.

Other good textbooks include:

- "Computer Security"(Gollmann, 2010). The third edition is most recent, but any edition is fine. It has more material on computer security aspects, and models of security, than any of the other books, and less on ciphers and hashes. A good complement to the others. Published in 2010. From about £30.
- "Cryptography Engineering – Design Principles and Practical Applications" (Ferguson et al., 2010). Published in 2010, this is an updated version of a book called "Practical Cryptography" by two of the same authors, but the older book is sketchy on details and rather outdated. Alternates between chatty prose and detailed implementations of security primitives, but good on the common pitfalls. Cheap (from around £20).
- "Handbook of Applied Cryptography"(Menezes et al., 1996). As the title suggests, focuses on cryptographic topics, but does contain plenty of material on protocols. A comprehensive tome which contains the state of the art as of 1996, thus sometimes out of date. The entire book can be downloaded for **free** at `http://www.cacr.math.uwaterloo.ca/hac/`.

At the end of each chapter is a bibliography with references to relevant literature. Where possible, links to freely downloadable copies of the papers have been included. The internet, of course, contains copious material on computer security but beware: not everything on it is completely reliable.

### Course Website

The course page is at `http://www.cs.ox.ac.uk/teaching/courses/security/`. At the appropriate time the class exercises, practical manuals, lecture slides and lecture notes, will

all appear on the "course materials" page. This will also contain links to key items of security literature.

Please send any lecture note and problem sheet corrections to `adk@cs.ox.ac.uk`; an errata will appear on the course website if necessary.

## 1.2 Notation

Computer security is a broad discipline and different parts of it have different notational conventions. Some have no conventions at all. In these notes, tuples will be written $\langle a, b\rangle$ and sequences $\langle x_1, x_2, \ldots, x_n\rangle$. When applying a function to a single sequence the argument will be written without parentheses, thus $f\langle x, y\rangle$ instead of $f(\langle x, y\rangle)$. The operator $\oplus$ means exclusive-or (XOR), applied elementwise if to sequences of bits and bitwise if to words.

Concatenation is written $\|$ (a computer security standard) and this notation is abused to mean both the formation of concatenated sequences and their deconstruction, thus $x = y \,\|\, z$ splits $x$ into $y$ and $z$. We will not always mention *how* the algorithm involved is supposed to know where $y$ ends and $z$ begins, but it will usually be either because they are of fixed length or because there is an unambiguous separator. We will be very lazy about types: an integer $n$ will be used interchangeably with the sequence of bits representing $n$ (but one has to be careful about this in practice: see endian-ness).

If $\Sigma$ is an alphabet (in the course usually either binary digits or bytes) then $\Sigma^n$ denotes the set of sequences of length $n$, and $\Sigma^*$ the set of all finite sequences in $\Sigma$. The zero-length sequence will be denoted $\langle\rangle$.

In most of the course, $x \pmod n$ denotes the nonnegative remainder when $x$ is divided by $n$. In chapter 5 we will use it in the more mathematically precise sense.

## 1.3 Aspects of Computer Security

Computer security is an extremely wide field, and difficult to define. It includes purely mathematical topics such as cryptography, and abstract quantifications of cryptographic security, through to rather non-technical subjects such as access policy and resource allocation. Computer security is primarily concerned with information flow, and some people define computer security as that subset of information security which pertains to

computers. In this course we stick mainly to matters of network communications security (chapter 2 is the exception).

Security means freedom from risk or danger: thus we define security by the risks and dangers we want to avoid. In computer systems, these risks include unavailability of a system or unauthorized behaviour by users; in communications systems they include unauthorized eavesdropping, tampering, or redirection of messages. It includes both prevention and detection. We restrict our attention to **malicious** behaviour by so called **attackers**, leaving computer reliability to hardware experts and communications fidelity to engineers.

There are a number of particular aspects of security which we want to consider here. We will look at each in turn.

### 1.3.1 Authorization

Authorization specifies the rights of actors to access resources. This includes the rights to view or change information on a shared system or database, as well as rights to know or alter the content of certain communications. It is the most basic element of computer security, as the policies which circumscribe these rights also define the security threats. The word **attacker** is synonymous with **unauthorized actor**.

It is normally policed by the operating system, database management service, or other program which administers the information. (Ensuring authorization in the absence of a so-called reference monitor is particularly challenging.) The main difficulty with authorization is less often about ensuring that the policies are followed, than in describing and maintaining them correctly.

### 1.3.2 Confidentiality

Confidentiality means that information is not disclosed to unauthorized entities. It is sometimes referred to as **secrecy** or **privacy**, both imprecise terms which can have other meanings. In terms of information flow it is a safety property: information does not flow from authorized to unauthorized entities.

Confidentiality is necessary for private data (medical records, financial information), private communication (diplomatic messages, sensitive emails), or even for storing authorization tokens. Breaches of confidentiality can happen if computer media is mislaid, if

internet packets are intercepted or radio overheard, or if an unauthorized user plays tricks on a computer system to deduce contents of files to which they should have no access.

It is typically achieved using symmetric or asymmetric encryption (cryptography), which are the topics of chapters 3 and 5. Note that confidentiality does not mean that the *existence* of the private information is kept secret, only its contents.

### 1.3.3 Integrity

Integrity means that *if* information is altered by unauthorized entities *then* authorized entities are aware that it was altered. Some authors consider that integrity means that unauthorized alteration is impossible, but this is not realistic for unsecured communication channels. Instead, we say that the receiver (or reader) of altered information will refuse to accept it, if unauthorized alteration is detected.

Examples of where integrity is necessary include any communication or stored data which intrinsically provides authorization (to avoid bypassing the authorization), financial transactions (to avoid an unauthorized party diverting funds), or safety critical information.

It is typically achieved by including, along with information whose integrity is to be assured, a cryptographic hash function (chapter 4) of that information. It can later be checked that the information still maps to the same hash value, and it is practically impossible for an attacker to alter the information without breaking this relationship. The hash will either have to be encrypted or **keyed** (first part of chapter 6). It is a common error to think that cryptography alone automatically ensures the integrity of confidential communications.

### 1.3.4 Authentication

Authentication refers to the verification of identity. It includes authentication of an entity to a computing resource ("logging on" by password or suchlike) and authentication of one entity to another (verifying other users' identities remotely, usually in the context of verifying the originator of a message). It is a safety property of information flow in the sense that an attacker is unable to spoof an identity.

Authentication is necessary from the very beginning, since users must identify themselves to their machines before they can hope to identify themselves to anyone else. (This mode of authentication should not be confused with authorization: possession of a password does not necessarily imply authorization, if the password was stolen.) It is important

in all non-anonymous, non-broadcast, communication as there is little point ensuring confidentiality and integrity of communications if they are going to the wrong place.

Entity to entity authentication is typically achieved using asymmetric cryptography, and a trusted third party to supply or verify the cryptographic keys of the entities involved.

When discussing communication channels, authentication has a particular meaning. A channel is authenticated if every message on it

(i) was sent by the purported sender
(ii) who intended to send it to the recipient, and
(iii) has not been received out-of-order or in duplicate.

Integrity is usually implicit in this sense of the authentication.

### 1.3.5 Non-repudiation

Non-repudiation means that an actor cannot deny having taken a particular action. In this course it will be limited to an actor being unable to deny that they authored a particular message. It means that unforgeable evidence exists that the message was sent by them. More widely, one can separate non-repudiation of sending from non-repudiation of receipt; we will only consider the former. It is a liveness property of information flow, and a sort of complement to **anonymity** (which does not appear in this course).

Non-repudiation is necessary if digital communications are to be binding in the same way that conventional signatures are. It is fairly worthless without integrity and authenticity, of course. Non-repudiation is typically achieved using **digital signatures**, which involve asymmetric cryptography and cryptographic hash functions. They provide a combination of integrity, authenticity, and non-repudiation.

### 1.3.6 Availability

Availability means that information and resources are available to authorized parties. In terms of information flow it is a liveness property: information does flow when authorized. In this sense it is a complement to authorization.

The necessity of availability is clear: without its requirement, the most secure computer system is that which is switched off. Availability may be breached by denial of service attacks, which are becoming an increasingly important part of computer security. But they are not within the scope of this course.

### 1.3.7 Other Aspects

There are many other aspects of security which we do not have time to cover here. They include **accountability** and **audit** (positive information), **anonymity** and **plausible deniability** (negative information). Note that these aspects can be contradictory because different information flows are desirable in different situations: anonymity is a must for electronic voting systems, whereas accountability is vital for payment systems. Another security aspect, perhaps less familiar, is **commitment**, where an actor reveals information which limits one of their later actions.

## Bibliography

Ferguson, N., Schneier, B., & Kohno, T. (2010). *Cryptography Engineering – Design Principles and Practical Applications.* Wiley.

Gollmann, D. (2010). *Computer Security.* Wiley, 3rd edition.

Kaufman, C., Perlman, R., & Speciner, M. (2002). *Network Security: Private Communication in a Public World.* Prentice Hall, 2nd edition.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography.* CRC.

# Chapter 2

# Access Control

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §1.8, 1.13 (barely covered) |
| **Alternatives & further depth:** | Gollmann chapters 5–7, 11 (goes way beyond the syllabus)<br>Ferguson et al.: no significant material<br>Menezes et al.: no significant material |

In this chapter we are concerned with **authorization**, when a computer system limits access to resources. First we consider how to model the access rules, second how to model security of a system which implements the rules, and finally how systems enforce the rules in practice.

This is a large topic and the chapter forms only a brief survey. Some computer security books do not discuss this much, and there is little in the course text (Kaufman et al., 2002). The book (Gollmann, 2010) is a better source for this material, and goes much further than these notes.

## 2.1 Modelling Access Control

A **computer system** might be one multi-user machine, or a collection of computers and other devices offering some sort of distributed service. In this chapter we assume that all users are who they purport to be (i.e. authenticity has already been established).

The terminology of access control is as follows. A **subject** is an entity[1] who wishes to access a certain **object**, which is some kind of resource: a file or directory, printer or webcam, or even a table within a database. There are usually different **modes** of access: reading, writing, or executing a file; listing, changing path to, adding to, or deleting from a directory; reading or writing a peripheral or network socket; selecting from, inserting into, deleting from, or dropping a table in a database (this list is not exhaustive). These modes are called **permissions**.

In models of access control, the modes are usually given the same set of names for all resources (e.g. "read", "write", "execute", "append") with each being interpreted differently on different types of resource[2]. Let $\mathcal{S}$ be the set of all subjects, $\mathcal{O}$ the set of all objects, and $\mathcal{P}$ the set of all permissions.

The complete description of access control can be given by a set $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{O} \times \mathcal{P}$. If the triple $\langle s, o, p \rangle \in \mathcal{A}$, then subject $s$ has permission $p$ to object $o$. When a subject requests a particular permission (or set of permissions) for a particular object (or set of objects), their request is granted or refused depending on whether their request is a subset of $\mathcal{A}$. Thus there are two sets: those of permissions which will be granted if requested, and those which are currently granted. When new permissions are added, triplets are added to $\mathcal{A}$; when they are removed (**revoked**), triplets are deleted.

A set of triplets is an ugly structure to reason about, so the literature uses a number of more refined representations of the same information:

- An **access control matrix** is a matrix $(M_{s,o})$ whose rows are subjects and columns are objects. Element $M_{s,o} \subseteq \mathcal{P}$ is the set of permissions that subject $s$ is authorized for object $o$.
- An **access control list** is a set $\{A_o \mid o \in \mathcal{O}\}$, one element for each object. The elements of the list are the pairs $\langle s, p \rangle$ of subjects $s$ who have permission $p$ to that object.
- Storing **capabilities** means giving to each subject **tokens** which give them access to the permissions they are entitled. Tokens are generated by the access control system, and tested at the point of use. They are stored by the subject.

[1] Subject is not exactly synonymous with **user**, because a user might have multiple identities on the computer system, corresponding to different roles.

[2] In practice, a persistent difficulty is that the exact meaning of permissions can be misunderstood. For example, permission to execute a file may or may not imply permission to read it, and "execute" permission on a directory is an unintuitive concept.

In abstract, access control lists (ACLs) store access control information by object, which makes it easy to tell which subjects have permissions to a particular object but difficult to list all the permissions of an individual subject. Conversely, capabilities store access control information by subject, which makes it easy to list the permissions of each subject but difficult to find all the subjects with access to a particular object. Furthermore, capabilities are stored by the subject rather than the access control system (they are generated and checked by mechanisms with which we will not concern ourselves). They have the advantage that they can be transmitted over a network (if securely!) and used in a distributed system.

Similarly, ACLs make it easy to grant or revoke permissions for a particular object, for example to make a file secret or to begin allowing access to a network resource: only one list need be updated. Capabilities make it difficult to revoke permissions, since the tokens are stored by the subjects: various mechanisms exist, including timeouts and one-time use.

An important distinction is between **mandatory** and **discretionary** access control. In the former, there are system-wide policies which determine the permissions, and only the security policy administrator can change them. In the latter, each object is assigned an **owner**, who is allowed to grant or revoke permissions for that object. Most familiar operating systems and database management systems operate discretionary access control, but some special versions (particularly for military or medical use) enforce mandatory controls, often with some limited discretionary controls as well. The advantage of mandatory access controls is that the security of the system does not depend on the good behaviour of users.

## 2.2 Modelling Security

A **security model** is a mathematical formulation of security policies. An implementation of an access control system (e.g. an OS) can be tested against the model. For example, we can ask whether the system protects the confidentiality or integrity of certain data. In military settings, it may even be necessary to supply a formal proof that the implementation meets the requirements of the model. There is much that could be said about security models, but most will not be said here.

Perhaps the most famous security model is the **Bell-LaPadula** (BLP) model, which dates from the 1970s. A nicely readable exposition is given in (Rushby, 1986). This

model aims to capture confidentiality requirements only; richer models are necessary to specify integrity and other security properties. At its heart is a **multi-level** security policy: there are a set of **security levels** $\mathcal{L}$, which are comparable by an operation $\leq$[3]. Then functions $f_S : \mathcal{S} \to \mathcal{L}$, which assigns to every subject their maximal security level or **clearance**, $f_C : \mathcal{S} \to \mathcal{L}$, which assigns to every subject their *current* security level, and $f_O : \mathcal{O} \to \mathcal{L}$, which assigns to every object a security level known as its **classification**.

The BLP model considers four modes of permission: READ access, WRITE access *which implicitly includes the ability to observe the object being written to*, APPEND access which allows blind writing but no reading, and EXECUTE permission which, by itself, permits neither read nor write.

The idea of **state** is important in the BLP model: the system is modelled as transitions through a set of states, starting from an initial state. The transitions are operations which change users' current or maximal security level, change the security level of objects, grant or revoke any kind of access, and so on. The security policy leads to a set of conditions on the states. The one relevant to discretionary access control is simple:

> A state satisfies the **ds-property**[4] if, whenever access $\langle s, o, p \rangle$ has been granted, $\langle s, o, p \rangle \in \mathcal{A}$.

The mandatory access control properties, which prevent the leakage of high-classification data to low-clearance users, are more difficult to get right. The first says that no subject may observe (read or write) any object with a higher classification. This is known as a **no read-up condition**:

> A state satisfies the **ss-property**[5] if, whenever access $\langle s, o, \text{READ} \rangle$ or $\langle s, o, \text{WRITE} \rangle$ has been granted, $f_O(o) \leq f_S(s)$.

This alone does not prevent information from flowing from areas of high-clearance to low-clearance, because a high-clearance subject would be allowed to WRITE or APPEND information into a low-classification object. The second property states this should not

[3] The formal requirement is that $L$ should be a lattice, though it is not uncommon for $\mathcal{L}$ simply to be an ordered list e.g. UNCLASSIFIED $\leq$ CONFIDENTIAL $\leq$ SECRET $\leq$ TOP SECRET.

[4] The "discretionary security" property.

[5] The "simple security" property.

happen, a **no write-down condition**. But high-clearance users can temporarily reduce their level (recall the distinction between maximal and current security levels), in which case they should not be able to read above the clearance of the object they are writing to:

> A state satisfies the **$*$-property**[6] if, whenever access $\langle s, o, \text{APPEND}\rangle$ or $\langle s, o, \text{WRITE}\rangle$ has been granted, $f_C(s) \leq f_O(o)$ **and** $f_O(o') \leq f_O(o)$ for all objects $o'$ where $\langle s, o', \text{READ}\rangle$ or $\langle s, o', \text{WRITE}\rangle$ has been granted.

Some versions also include the so-called **tranquility property** which prevents objects' classifications from being changed while they are being accessed.

The general idea is that an implementation is checked against this model, proving that the initial state satisfies the security properties and that every implemented transition preserves the properties. In which case, the system is provably secure at all times. The BLP model, however, is inadequate to deal with issues more complex than confidentiality and one would look to models by Biba or Clark/Wilson. Another issue is the change of permissions, for which the BLP model allows great latitude.

## 2.3 Enforcing Access Control

Access control is implemented in the operating system (or web server, database management system, etc.) which runs on a hardware device. The part of the OS which decides whether each access is permitted is called the **reference monitor** and it acts as a gate through which all accesses are made. (The OS will most likely also enforce audit, i.e. logging of certain actions.) But software alone is not enough to enforce access control. Consider a process running under an operating system: if the process were allowed to perform any processor instruction it wished, it could issue instructions to overwrite the operating system itself, disabling the reference monitor. The first principle of access control is that the reference monitor's integrity must be guaranteed. Thus enforcement of access control must begin in the hardware itself.

Modern processors support a notion of security level. This idea was introduced by Multics, which used eight levels, and modern x86 processors support four levels. In most operating systems, only two levels are used: a **user mode** (called ring 3 on x86 processors) and

[6] Apparently so-called because the authors could not think of a name and used $*$ as a placeholder.

a **kernel mode** or **system mode** (ring 0). Each thread of execution, and each page of memory, is marked with a level. Threads running in user mode are refused access to certain processor instructions. Prohibited instructions include access to pages of memory owned by kernel mode threads, instructions to access hardware directly, and instructions to modify system interrupts. (There is no prohibition on kernel mode threads writing into user mode pages.) The prohibitions are enforced by a little reference monitor built into the processor, operating at the silicon or microcode level. So underneath the OS access control is a simple multilevel access control system inside the CPU.

When an operating system starts, almost its first task is to request access to kernel mode. The part of the operating system which will deal with hardware (in monolithic kernels this means all device drivers), and also the OS reference monitor, run in kernel mode, and user-generated processes run in user mode. Thus processes spawned on the OS cannot attack the OS reference monitor[7]. User mode processes have means to request kernel mode processes to perform actions on their behalf, but their authorization is checked first and the so-called context switch is a notoriously slow operation, which is why more and more modules make their way into kernel mode.

Thus protected by the hardware, the operating system can authenticate users (who log in by password or other means that we will discuss later), keeping track of their actions and the processes they run. When a user starts a process, it runs with their privileges. A tricky problem arises when the following timeline occurs: the user requests access to a resource, which is granted; a change of authorization occurs revoking that access; the user performs the access. This is a race condition called **time of check to time of use** (TOCTTOU) and it seems to be difficult to solve.

Most operating systems include a special user called the **superuser**, often with the name **root**. The superuser is allowed universal access. Such a user is needed in order to change authorizations, and only they are allowed, by the OS, to perform certain actions including accessing privileged network ports and direct hardware interaction. Typically it is system administrators who know the root password, and it must be guarded carefully. It is important to note, however, that the superuser is still a user: their processes run in user mode and they have no special status in the CPU.

Let us briefly mention how operating system access control can be attacked. The aim of the attacker is to gain a permission to a resource not granted to them. There are

[7] But this is no good if an attacker gets to kernel mode before the OS does, as for example boot sector viruses do. On the other hand, wider use of processor rings is regaining interest as it can be used for virtualization, where the OS itself runs inside a hypervisor.

(very broadly) two approaches to this: tricking the reference monitor into granting an access it should not, or tricking the processor into executing some code which will alter or disable the reference monitor. Exploiting the TOCTTOU problem, where a user knows that extra permissions will briefly appear (perhaps while one file is being copied on top of another) and targets their authorization request to the window of opportunity, is an example of the former. Because of the hardware protections against user mode threads writing to kernel mode memory, the latter requires finding bugs in parts of the OS running in kernel mode.

Such bugs are common. Failure to enforce boundary conditions, to fully check syntax of user-supplied arguments, and allowing the infamous **buffer overrun**[8], can all result in user-supplied input to kernel mode threads overwriting the thread code itself. Although it might seem impossible to ensure that spewing partially-controlled data all over a kernel memory page could do anything other than crash the OS, hackers have found ways to control the outcomes and write precisely into the part of the OS which controls their access rights, giving them root access or other privileges. This is known as **privilege escalation**. It explains why it is dangerous when *any* user or network service in a system, no matter how restricted their own privileges, allows unauthorized access: a hacker may take advantage of the weak user and then perform a privilege escalation attack.

These methods are not only for evil hackers. Many people own mobile phones with very restrictive access controls, and they wish to make more use of their device – which they legally own – than the manufacturer intended. So-called **rooting** or **jailbreaking** of a device means attempting to gain superuser access on the underlying operating system, and it typically involves exploiting a bug in the kernel mode code (of course, the manufacturer updates the code so that the same bug does not exist in the next version of firmware). The protection mechanisms for modern phones – and modern games consoles – are really quite complicated, involving multiple stages of security checks with a highly-secure bootloader which performs integrity checks on the operating system (see chapter 6) and refuses to boot tampered code.

Finally, there are ways attackers may gain access even to a perfectly secure system if the humans involved act insecurely. An attacker might fool another user into giving them access to something unwittingly (less successful in mandatory access control systems), or fool a superuser into changing their permissions. One of the most effective attacks

[8] A classic reference is found at `http://www.phrack.org/issues/49/14.html`, which explains the general idea of supplying long arguments to write into memory not intended for data.

on computer systems, certainly up to the end of the 1990s, was to phone up the system administrator and pretend to be another user, asking for a password reset.

## 2.4 Case Study: UNIX Permissions

Time permitting, the lectures will discuss how traditional UNIX/POSIX systems implement file and directory access control. There is such copious information online and in books like (Barrett et al., 2003) that it would be pointless to include it again in these notes. Only the bare bones appear here.

The permissions are closer to ACLs than capabilities, in that they attach to the files themselves, but they are not as fined-grained as full ACLs[9]. This is to allow for more efficient administration. Each user (subject) has an integer identifier (UID) and belongs to at least one **group**. Groups are also identified by an integer (GID). Each file is "owned" by a single user and a single group, and has nine permissions: read (`r`), write (`w`), and execute (`x`) for its owner, members of its group, and everyone else. Every process runs with a UID and GID, normally those of the user who executed it. If a process $p$ tries to access an object (file or directory) $o$, the OS: checks whether $p$ has the same UID as the user owning $o$, in which case it determines the permissions based on the user `rwx` flags; if not, it checks whether $p$ has the same GID as the group owning $o$, in which case the group `rwx` permissions are used; if not, the remaining `rwx` permissions are used.

Only the owner of a file can change its permissions, except that the superuser `root` has universal rights to access files and change permissions (and owners, groups, etc).

The group system makes it easy to add new users whose access rights are similar to others', or to change access permissions en masse, if the groups correspond well with users **roles**. But, like all access control systems, it has a number of quirks which make it easy to misconfigure the permissions: if a file's owner is also in the group which owns it, they do not inherit the rights of the group; execution of certain files will not work unless read permission is also set (but for certain files it will); permissions on symbolic links seem to have no useful meaning. Finally, permissions for directories contain many traps: read permission determines whether the names of its files can be listed, but execute permission allows the files to be used if accessed directly by their names. And write access allows any file in the directory to be removed, regardless of the access permissions of the file itself!

[9]Modern linux filesystems do support ACLs, but they are usually disabled.

The first lesson of any access control system is to make sure that you fully understand the meanings of the permissions, before attempting to configure the rights of the subjects.

## Bibliography

BARRETT, D. J., SILVERMAN, R. E., & BYRNES, R. G. (2003). *Linux Security Cookbook*. O'Reilly.

RUSHBY, J. (1986). The Bell and La Padula security model. Technical note. SRI International Computer Science Laboratory. Available at `http://www.csl.sri.com/users/rushby/papers/blp86.pdf`.

# Chapter 3

# Symmetric Key Ciphers

---

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §2.1-2.2, 3.1-3.3, 3.5-3.6, 4.2, 4.4 |
| **Alternatives & further depth:** | Gollmann §14.5, 14.5.1, 14.5.2, 14.6 (very sketchy)<br>Ferguson et al. §2.6, 2.7.2, 2.8, chapters 3 & 4<br>Menezes et al. §7.1-7.4, 1.13.1, 1.13.3, 6.1 |

---

In this chapter we consider ways to achieve **confidentiality**. We have in mind the following scenario: Alice, a sender, wants to transmit a message to Bob, the receiver. Unfortunately, the channel over which she transmits is insecure, perhaps involving radio (which might be received by any party) or the internet (on which any router can read packets passing through). The message should be kept confidential, despite the possible presence of an **eavesdropper** who is called the **attacker**, **enemy**, or **cryptanalyst**. In the first part of this course we will mainly consider this so-called **passive intruder**, who is able to eavesdrop, but not to alter the content of messages or redirect them completely.

This same scenario also covers secure storage, if we imagine that Bob is the future version of Alice and the transmission channel is the storage medium.

The classical way to achieve confidentiality is to use cryptography: Alice encrypts the message using a secret key which she shares with Bob. In this chapter we will not worry about how Alice and Bob come to share the key: that is a problem we shall address later.

## 3.1 Formal Definition

Formally, **encryption** and **decryption** are functions. We suppose that there is a set $\mathcal{M}$ of messages or **plaintexts**, which is mapped to a set $\mathcal{C}$ of **ciphertexts**. The encryption **key** comes from a set $\mathcal{K}$ called the **keyspace**. (In practice, usually both $\mathcal{M}$ and $\mathcal{C}$ are the sets binary strings of fixed length $n$, where $n$ is called the **block size**. $\mathcal{K}$ is often, but not always, a set of binary strings of length $k$, the **key size**.) Then we have encryption and decryption functions

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}; \quad D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}.$$

Some people just write $E(k, m) = c$ but most write the key as a subscript, thus $E_k(m) = c$ means that message $m$ is transformed into ciphertext $c$. The sets $\mathcal{M}$, $\mathcal{C}$, and $\mathcal{K}$, together with $E_-(-)$ and $D_-(-)$ are called a **cryptosystem** or **cipher**[1].

We require that messages are passed correctly by the cryptosystem, thus

$$\forall m \in \mathcal{M}.\ D_k(E_k(m)) = m.$$

The above definitions formalise a **symmetric key cryptosystem**, so called because the same key is used for encryption and decryption. In chapter 5 we will meet asymmetric key cryptosystems, in which the decryption key differs from the encryption key.

That decryption correctly reverses encryption is, of course, only part of the requirement for a cryptosystem. We also need to know that an enemy with access to ciphertexts cannot deduce the plaintexts without knowledge of the key. This is much more difficult to formalise (in fact to formalise completely would be beyond the scope of this course). We must first know what kind of enemy we are working against.

## 3.2 Attacks

An enemy, or eavesdropper, is defined by what they know about plaintexts and ciphertexts. There are many different **attack models** in cryptography.

[1] Strictly speaking, the cryptosystem should also include the method for generating keys. We postpone this for later.

The first thing we must admit is that the attacker knows the encryption and decryption functions. This is known as **Kerckhoffs' Principle**, named after a Dutch cryptographer who wrote a famous treatise in the 19th century enumerating some properties of a proper cryptosystem. Claude Shannon translated the Principle as "assume the enemy knows the system". It is simply too risky to try to keep the encryption and decryption functions secret, because there might now or one day be a traitor in your communication system, or a piece of encryption software might fall into enemy hands. Furthermore, it is presumably a lot cheaper to change an encryption key, should it become disclosed to the enemy, than to change the entire cipher.

This is related to the debate on "security through obscurity" in software design, but in cryptography there is no debate: a cipher which relies on the secrecy of its operation is not secure.

### Ciphertext Only Attack (COA)

In this attack, the enemy only has access to a number of ciphertexts. Unless the cipher is completely broken it will be impossible for them to say anything about the corresponding plaintexts unless they have *some* sort of information about possible messages. But in practice the enemy may well have such information:

- They might know that the plaintext consists of English words encoded, for example, in ASCII.
- They might at least know that the plaintext consists of ASCII characters encoded as bytes (this is still powerful information: in plain ASCII, the most significant bit of every byte is zero).
- They might know that the plaintext carries an email, in which case it will have recognisable headers.
- They might know that the plaintext has been compressed using off-the-shelf compression software, in which case the first few bytes will be a recognisable header.
- Similarly, they might know that the plaintext is a JPEG or TIFF image (recognisable headers), an mp3 file (ditto), a piece of video (ditto), an binary file executable on some operating system (ditto), . . . .

This is known as the **recognisable plaintext** assumption and it is necessary for a COA attack to be possible. Nonetheless, the COA is the most difficult attack to succeed with, and therefore makes for the weakest enemy.

### Known Plaintext Attack (KPA)

Here we assume that the enemy knows the encryption of a certain plaintext, or knows a limited number of plaintext-ciphertext pairs, all under the same key. This is a plausible assumption if secrecy is lifted after a certain period of time. Or the contents of the plaintext might be deduced by other means, for example in the case of the Enigma machine where the Allies realised that a weather report was always sent at a certain time of day.

The attacker attempts to deduce the key, or the decryption of new ciphertexts (encrypted with the same key) from the known plaintext-ciphertext pairs. This attack is easier than the COA, and most weak cryptosystems fall to it very quickly.

### Chosen Plaintext Attack (CPA)

The attacker is able to find out the encryption of *any* plaintext they want. Usually the attacker is allowed to do this as many times as their computation power allows. At first sight this might seem unreasonable, but consider that there might be a traitor somewhere in the communication system who can plant messages which will eventually be transmitted by Alice. Or the attacker might be able to provoke Alice into forwarding a certain message, along the lines of "please tell Bob..." (this was used in attacking Enigma). Chosen plaintexts are completely normal in public key cryptography, which we shall come to in chapter 5, because the encryption key is made public.

There are two variants of the CPA. In the plain version, the attacker is supposed to decide on the plaintexts they choose to have encrypted all at once; in the *adaptive* version, they are allowed to use the results of previous encryptions to choose future plaintexts.

### Chosen Ciphertext Attack (CCA)

The toughest attack model gives the attacker the power to *decrypt* chosen ciphertexts even though they do not know the key. The only restriction is that they may not decrypt the one ciphertext which they are trying to break. This is realistic in certain public key scenarios (we shall see in chapter 6 that digital signatures are related to decryption), or in circumstances where a user leaves a computer unattended during which time, it is hypothesised, the attacker borrows it to make some decryptions in the hope of deducing the key.

Like the CPA, there are two variants of the CCA depending on whether the attacker is allowed to choose the decryptions adaptively or not.

A cipher is considered **secure** only if it is secure against at least the CPA (in the sense that it requires the attacker to perform an unfeasible amount of computation). For some situations, particularly with public keys, security against CCA is also required. No matter how proud one might be of a new cipher, if it fails to respect Kerckhoffs' Principle or falls to a COA, KPA, or CPA, it is not secure.

## 3.3 Weak Ciphers

There is copious non-academic literature on code breaking, which contains many examples of weak ciphers, e.g. books by Simon Singh, David Kahn, Bruce Schneier, etc. This section therefore contains just a few examples to illustrate important principles in the design of good ciphers.

If the keyspace is too small, the cipher is certain to be weak. A classic (some would say hackneyed) example is the **Caesar cipher**. The plaintexts and ciphertexts are fixed-length blocks of Roman alphabet characters (no spaces) encoded as integers $A = 0$, $B = 1, \ldots, Z = 25$, thus $\mathcal{M} = \mathcal{C} = \{0, \ldots, 25\}^n$. The key is an integer in the range $0, \ldots, 25$. The encryption and decryption functions are cyclic shifts on each character[2]:

$$E_k\langle m_1, \ldots, m_n\rangle = \langle m_1 + k, \ldots, m_n + k\rangle \pmod{26},$$
$$D_k\langle c_1, \ldots, c_n\rangle = \langle c_1 - k, \ldots, c_n - k\rangle \pmod{26}.$$

Then `caesarcipher` encrypts under key $k = 10$ to `mkockbmszrob`. Even without knowing the decryption key you can probably work out for yourself what `nbcmcmuqyuewcjbyl` was the ciphertext for.

The Caesar cipher is weak for many reasons, one of which is the very small keyspace. It takes but an instant for a computer to try all 26 possible keys to determine which is the decryption of plausible English (a ciphertext only attack) or which matches a single known plaintext-ciphertext pair (a known plaintext attack). The keyspace must always be large enough to protect against this **exhaustion attack** (also known as the **brute force attack**) which tries all possible keys: in the worst case the exhaustion attack requires $|\mathcal{K}|$ attempts to find the right key, but on average only $\frac{1}{2}|\mathcal{K}|$. The minimum size keyspace to make exhaustion infeasible depends on how long it takes to try a decryption, but in practice $2^{40}$ keys is much too short, $2^{64}$ is nowadays still too short[3], $2^{80}$ is good

[2] This is effectively ECB on a single-character cipher, see section 3.7.

[3] The first publicly-known exhaustion attack against a 64-bit key was completed in 2002, taking 5 years and using the spare computer time of tens of thousands of computers `http://www.distributed.net/RC5`. But 64-bit exhaustions are now considered practical for a determined opponent.

for security right now, but we would want at least $2^{96}$ if preserving secrets for 30 years. Bear in mind that an enemy might store our ciphertexts for a long time, and that for serious secrets we might want to maintain confidentiality for more than 30 years.

A cipher which *looks* stronger than the Caesar cipher is the **monoalphabetic substitution cipher**. With the same encoding of blocks of letters as above, the keyspace is any permutation of the numbers $0, \ldots, 25$, and the encryption and decryption functions just apply the permutation to each letter:

$$E_\pi\langle m_1, \ldots, m_n\rangle = \langle \pi(m_1), \ldots, \pi(m_n)\rangle, \quad D_\pi\langle c_1, \ldots, c_n\rangle = \langle \pi^{-1}(c_1), \ldots, \pi^{-1}(c_n)\rangle.$$

In plain language, every letter of the alphabet is substituted by another. Of course, the Caesar cipher is just a special case of the monoalphabetic substitution cipher, but now the keyspace is considerably larger. There are $26! \approx 2^{88}$ possible permutations[4], which should certainly defeat a casual exhaustion attack, but it is still insecure because it preserves the structure of the plaintext, in this case the English language. Such ciphers have long been broken by **frequency analysis** (knowledge that certain letters occur more often than others) and other such tricks. You may wish to try your hand at deciphering `pdttrbadwblpvaabchvzd` (and then working out where the key came from). And of course a single known plaintext-ciphertext pair is also enough to break the code.

Frequency analysis is an example of **cryptanalysis**, the aim of which is to break the confidentiality of the ciphertext. A cryptanalysis is considered successful if it requires less work than an exhaustion attack on the same cipher.

Let us turn to more realistic computer ciphers, based on blocks of binary. Suppose that the plaintexts, ciphertexts, and keys are sequences of $n$-bits $\{0,1\}^n$. For reasonably large $n$, the keyspace is much too big for exhaustion attacks. We define

$$E_k(m) = m \oplus k, \quad D_k(c) = c \oplus k,$$

where $\oplus$ represents the bitwise exclusive-or operation (XOR). Recall that XOR is a self-inverse operation. This is the Vernam cipher, of which more shortly. It is "perfect" in a theoretical sense we will define in the next section, but completely insecure against a single known plaintext-ciphertext pair: given $E_k(m) = c$ we simply deduce

$$k = c \oplus m.$$

[4]The size of keyspace is usually measured in bits, so we would say that this cipher has an 88-bit keyspace.

The root of this weakness is that the cipher is **linear** in the key, making it simple to solve the equation $E_k(m) = c$ for $k$. There are other forms of linearity which can lead to weakness. For example, if we remove the linearity in $k$ by changing the cipher to

$$E_k(m) = m \oplus h(k), \quad D_k(c) = c \oplus h(k),$$

where $h$ is some one-way function which is practically impossible to invert (the topic of chapter 4) then one cannot solve $E_k(m) = c$ for $k$. But, given $m$ and $c$, we can still decrypt any other message using the same key, because $E_k(m') = c'$ implies that

$$c \oplus c' = m \oplus m',$$

from which $m' = c' \oplus m \oplus c$ recovers the unseen plaintext $m'$. This version of the cipher was still linear in $m$, causing the weakness.

The word linearity is used to mean a number of related things in cryptography, for example when $E_k(m_1 \oplus m_2)$ is some easy to compute function of $E_k(m_1)$ and $E_k(m_2)$, or $E_{(k_1 \oplus k_2)}(m)$ a simple function of $E_{k_1}(m)$ and $E_{k_2}(m)$. Linearity can occur with respect to normal addition or indeed any other operation, but XOR is the most commonly-exploited example. All types of linearity are dangerous; even without a known plaintext-ciphertext pair, linearity properties give rise to nonuniformities which can be exploited for COA. Some subtle contemporary attacks on difficult ciphers work by making linear approximations to cipher components.

Thus it is extremely important to include nonlinear operations in the design of good ciphers.

## 3.4 Perfect Security

As computer scientists, we would like to have a proper definition of "security" in the sense of confidentiality. The first definitions began to appear in the 1940s, beginning with the idea of **perfect security** or **perfect secrecy**. You can find many definitions of perfect security in the literature, all aiming to capture the idea that an attacker can say nothing new about the plaintext once given the ciphertext, and almost all of them turn out to be equivalent. Here we will go back to the first such definition, from Shannon's seminal postwar publication (Shannon, 1949).

### 3.4.1 Shannon Security

First, recall some discrete probability. For events $A$ and $B$, $\mathrm{P}[A]$ denotes the probability that $A$ occurs, $\mathrm{P}[A \wedge B]$ the probability that $A$ and $B$ both occur, and $\mathrm{P}[A \,|\, B]$ the **conditional probability** that $A$ occurs "given that $B$ occurs" formally defined by

$$\mathrm{P}[A \,|\, B] = \frac{\mathrm{P}[A \wedge B]}{\mathrm{P}[B]}.$$

A **random variable**, formally a function on a sample space, can be thought of as a random quantity. If $X$ is a random variable drawn from a set $\mathcal{X}$, then $X = x$ denotes the event that $X$ takes value $x$ (it is conventional to write random variables as uppercase letters, with lowercase letters for the corresponding realisations). A random variable $X$ is defined by its **distribution**, which is the probabilities of all the events $X = x$ for each $x \in \mathcal{X}$. The **uniform distribution** on $\mathcal{X}$ is the one where each $x$ has the same probability, $\mathrm{P}[X = x] = 1/|\mathcal{X}|$ for each $x$.

Shannon's notion of perfect security assumes that the messages are chosen at random from $\mathcal{M}$ with some (any) distribution, and similarly for the keys. This means that the ciphertexts are also random. Although we might think, as cryptographers, that our messages are certainly *not* random, it is plausible that the attacker is intercepting messages at random, or they can model us as a random process.

Shannon's definition of **perfect security** is:

(i) For all $m \in \mathcal{M}$ and $c \in \mathcal{C}$,

$$\mathrm{P}[M = m \,|\, C = c] = \mathrm{P}[M = m].$$

(ii) Also, the keys are drawn **uniformly** from $\mathcal{K}$, and each key is used for one encryption only.

(i) captures the idea that the attacker gains no extra information about the message $m$, if they are given the ciphertext $c$. They might as well not have seen $c$ at all. It corresponds to the idea of defeating a COA and does not make any assumptions about limitations on the attacker's computational power. (ii) makes a KPA, CPA, or CCA impossible, because the enemy will never see other encryptions or decryptions using the same key. Moreover, they have no way to guess which key is likely to be used next, because no key is more likely than any other.

Sadly for cryptographers, both conditions turn out to be impractical. It is easier to see why the second is difficult: it would be tiresome to change keys as frequently as every message. The first is also difficult because of the following theorem, which some call the **impracticability** theorem:

**Theorem 3.1** Let $\mathcal{M}' \subseteq \mathcal{M}$ denote the set of all messages with strictly positive probability. Shannon's first condition $\mathrm{P}[M = m \,|\, C = c] = \mathrm{P}[M = m]$ requires $|\mathcal{K}| \geq |\mathcal{M}'|$.

**Proof** First note that, as long as $\mathrm{P}[M = m] \neq 0$,

$$\begin{aligned} & \mathrm{P}[M = m \,|\, C = c] &=& \mathrm{P}[M = m] \\ \Rightarrow\ & \mathrm{P}[M = m \wedge C = c] &=& \mathrm{P}[M = m]\mathrm{P}[C = c] \quad \text{(hence } M \text{ and } C \text{ are } \textbf{independent}\text{)} \\ \Rightarrow\ & \mathrm{P}[C = c \,|\, M = m] &=& \mathrm{P}[C = c]. \end{aligned}$$

Now suppose that $|\mathcal{K}| < |\mathcal{M}'|$. Pick any possible ciphertext $c$ (i.e. something in the image of $E$). Because $c$ is possible, $\mathrm{P}[C = c] > 0$. Now consider the following subset of $\mathcal{M}'$:

$$\{m \,|\, D_k(c) = m \text{ for some } k\}.$$

Because the number of possible outputs of a function cannot be more than the number of possible inputs,

$$|\{m \,|\, D_k(c) = m \text{ for some } k\}| \leq |\mathcal{K}| < |\mathcal{M}'|.$$

This means that there exists at least one $m_0 \in \mathcal{M}$ with $\mathrm{P}[M = m_0] > 0$ and

$$D_k(c) \neq m_0 \text{ for any } k$$

which in turn implies that

$$E_k(m_0) \neq c \text{ for any } k$$

and hence that

$$\mathrm{P}[C = c \,|\, M = m_0] = 0.$$

Thus $\mathrm{P}[C = c] \neq \mathrm{P}[C = c \,|\, M = m_0]$, completing the contradiction. ■

Why does this make perfect security impractical? If there are at least as many possible keys as there are possible messages, it will take at least as many bits to store/transmit

the key as the message itself (this can be made more precise, using a more refined version of Shannon's theorem and the concept of **entropy**). And according to Shannon's second condition, we must share a new key for every message. Our assumption was that the communicating parties are able to share the secret key, but if the secret keys are longer than the messages then they might as well not bother with encryption and just share the message instead! The only advantage of perfect cryptography is that the keys can be shared well in advance.

### 3.4.2 The Vernam Cipher and the One-Time Pad

Notwithstanding the apparent difficulties, there does exist a perfect cipher which has been used in practice. It is based on the **Vernam cipher**, which you have already seen. It is named for its inventor Gilbert Vernam, who patented it in 1919. The message space, cipher space, and keyspace all equal the set of $n$-bit binary sequences $\{0,1\}^n$, with encryption and decryption given by

$$E_k(m) = m \oplus k; \quad D_k(c) = c \oplus k.$$

That is, the plaintext is simply XORed with the key. The Vernam cipher is insecure because it falls to a single known plaintext attack as we have seen. Even in the absence of a known plaintext, it can usually be broken by COA if the same key is used for more than one plaintext: an example along these lines appears on the first problem sheet.

Notwithstanding its insecurity,

**Claim 3.2** The Vernam cipher, with a truly random key (uniformly distributed over all $n$-bit sequences, independent of the message), meets Shannon's first condition.

**Proof** We use the independence and uniform distribution of the key to compute

$$\mathrm{P}[M = m \wedge C = c] = \mathrm{P}[M = m \wedge K = c \oplus m] = \mathrm{P}[M = m]\mathrm{P}[K = c \oplus m] = \mathrm{P}[M = m]2^{-n}$$

and

$$\mathrm{P}[C = c] = \sum_m \mathrm{P}[M = m \wedge C = c] = \sum_m \mathrm{P}[M = m]2^{-n} = 2^{-n}.$$

Dividing,

$$\mathrm{P}[M = m \,|\, C = c] = \mathrm{P}[M = m].$$

■

We can reconcile these apparently contradictory statements by remembering that perfect security requires that **the same key is never used twice** (strictly speaking, we should say that keys are re-used no more often than expected of uniformly randomly chosen keys). Thus the KPA does not apply, nor the COA requiring two ciphertexts with the same key.

When the Vernam cipher is used with uniformly random keys, changing the key for every encryption, it is called the **one-time pad** (OTP). Although it suffers from exactly the problem we described in the previous section – the secret key is the same size as the message being transmitted – it was certainly usable in practice. The KGB in the 1950s and 60s, for example, sometimes issued agents with pads of paper with pages covered in random letters. To communicate with their recipient, agents would form some type of XOR[5] of their plaintext with the text from the pad, and indicate at the top of the message which page of the pad was used. After encryption, they burned the page, and the recipient would burn their copy (the only other copy in existence) after decryption. If used properly this is perfectly secure.

In practice one needs to take great care in generating the key (the pads). Some diplomatic pads were generated by secretaries typing at random into actual typewriters with, presumably, carbon copy paper. People do not press typewriter keys uniformly at random! And a Soviet blunder resulted in the re-use of some OTP keys and this was exploited by the US and UK from the 1940s to the 1980s. Similar mistakes have caused the downfall of many ciphers, and history seems to show that the policy surrounding the use of cryptography is often more easily attackable then the cryptography itself.

## 3.5 Practical Security

Let us return to the real world of computer security. Perfect security seems to be impractical, so we must aim for near-perfect security. Defining near-perfect security is more difficult than defining perfect security, and anyway we care more about an attacker who has a limited amount of computation rather than, as in the last section, attackers with unbounded computational power. It took a while for computer scientists to come up with

[5]presumably modular addition.

a definition of a **computationally secure** encryption, and the technicalities are beyond the scope of this course[6].

We can make progress by going back to Shannon's 1949 paper and looking at his general comments on the design of good cryptography. Since we are working in the real world, from now on we will assume that the plaintexts and ciphertexts come from the same set of fixed-length sequences of bits $\{0,1\}^n$, and keys from $\{0,1\}^k$. Shannon gives two properties, slightly vague, which lead to good ciphers:

**Confusion:** Given $m$ and $c$, the equation "find $k$ such that $E_k(m) = c$" should be very hard to solve. This implies nonlinearity, amongst other things, since linear equations are easy to solve. The aim is to make the KPA difficult (and, since it should hold for all $m$, the CPA too).

**Diffusion:** Nonuniformity in parts of $m$ should be spread out so that $E_k(m)$, for any $k$, is close to uniform. This is designed to defeat a COA by obscuring redundancy (like the nonuniformity of letters in English).

Bear in mind that Shannon was thinking about cryptanalysis by hand, since this work was contemporaneous with only the very first computers, hence the idea that equations should be "hard to solve" by hand. Diffusion does have a more precise definition, but it is not helpful to us here.

As well as Shannon's insights, cryptographers tend to be guided by empirical evidence: if many academics have searched in vain for an attack on a particular cipher then it is probably secure. In (Kaufman et al., 2002) this is called the "Fundamental Tenet of Cryptography". In a field which is constantly evolving, with new types of attack discovered and new cipher styles to defeat them, this is about the best we can do.

### 3.5.1 Feistel Structures

It is not so difficult to invent an encryption function which has reasonable confusion and diffusion: just throw bits around, combine them in nonlinear ways, and repeat until you

[6]Definitions of a computationally secure encryption are usually based on the idea of a game, played between the sender and the cryptanalyst, where the cryptanalyst has to guess something about the sender's message given certain information. A simple example of such a definition would be as follows. Suppose that the plaintexts are uniformly distributed over $\mathcal{M}$. Then encryption is secure if, given randomly chosen $M$ and $K$, and for any function $A(x)$ computable in polynomial time, the difference in probability that $A$ gives the right decryption, compared to random guessing, $|\mathrm{P}[A(E_K(M)) = M] - \frac{1}{|\mathcal{M}|}| \leq \nu(\log|\mathcal{K}|)$, where $\nu(k)$ is some *negligible* function, one whose reciprocal grows more quickly than any polynomial.

think they are thoroughly mixed up. Unfortunately, there will probably be no decryption function to match it! The problem can be solved using a nice little trick, a structure called a **Feistel structure** or **Feistel network** (named after the IBM cryptographer who invented it).

Consider the following function on pairs of blocks:

$$\text{Round}(x, y) = \langle y,\, F(y) \oplus x \rangle.$$

were $F$ is some "mangler" function which mixes up its inputs in nonlinear and confusing ways. Although $F$ is probably not invertible, Round is invertible with inverse

$$\text{Round}^{-1}(x', y') = \langle F(x') \oplus y',\, x' \rangle.$$

Only the second part of Round's output is confused, of course, but the function can be iterated (still invertibly) to produce thoroughly mixed output[7].

A Feistel structure is a keyed version of this. We begin by splitting the plaintext $m$ (assumed to be a block of bits) into two halves $L_0$ and $R_0$ (conventionally these bit blocks seem to be written as uppercase letters). We require a function $F(K_i, X_i)$ which takes both a **subkey** or **round key** $K_i$ (part of the crypto key, usually a selection of bits from it) and bit block $X_i$. The aim of $F$ is to make the output confusing with respect to both $K_i$ and $X_i$. Then we iterate through $r$ **rounds**,

$$\begin{aligned} L_{n+1} &= R_n \\ R_{n+1} &= F(K_{n+1}, R_n) \oplus L_n \end{aligned}$$

Finally, $R_r \,\|\, L_r$ the ciphertext output (note the reversed order of blocks).

To decrypt a ciphertext $c$ we split into two halves $L'_0 \,\|\, R'_0$ and iterate

$$\begin{aligned} L'_{n+1} &= R'_n \\ R'_{n+1} &= F(K_{r-n}, R'_n) \oplus L'_n \end{aligned}$$

with $m = R'_r \,\|\, L'_r$. You can check (proof by induction) that this is the correct inverse. Swapping the order of the two halves of the ciphertext has given the handy property that

[7] There is some good theory to back this up: the Luby-Rackoff Theorem says that if $F$ is computationally indistinguishable from a random function then at least four iterations of the Feistel construction lead to an invertible function which is computationally secure.

decryption is the same as encryption, except that the subkeys are used in reverse order. So only one program, or digital circuit, needs to be built for both.

Many ciphers make use of some version of this structure, or variants of it (e.g. different mangler functions on each round, or unbalanced splitting into left and right halves).

### 3.5.2 DES

The **Data Encryption Standard** (DES) was the first cryptographic method certified by the US National Bureau of Standards, and it was became very widely used. It was a development of IBM's Lucifer cipher, and was developed by IBM's cryptographers in cooperation with the NSA[8], although the exact level of the NSA's involvement is disputed. The final version of DES was published in 1977[9]; it was not successfully attacked until the late 1990s.

DES uses 64-bit plaintext and ciphertext blocks and a 56-bit key. It consists of a 16-round Feistel network

$$L_{n+1} = R_n$$
$$R_{n+1} = F(K_{n+1}, R_n) \oplus L_n$$

where each subkey $K_n$ is a different selection of 48 bits from the key, and the mangle function $F$ can be concisely described by

$$F(K, R) = P(S(E(R) \oplus K))$$

where

$$E : \{0,1\}^{32} \rightarrow \{0,1\}^{48}$$

is the so-called **expansion permutation** which duplicates some of the 32 input bits to make the output up to 48 bits (it is not really a permutation),

$$S : \{0,1\}^{48} \rightarrow \{0,1\}^{32}$$

[8] The NSA is the government agency of the United States which is responsible for signals intelligence (i.e. eavesdropping) and communications security (i.e. cryptography). Its activities are classified and subject to much speculation. The UK equivalent is GCHQ.

[9] It was renewed in 1988 and 1993, the latest version of the standard is (National Bureau of Standards, 1993)

is a fixed nonlinear function. Its 48 bits of input are broken down into eight block of six, each of which is fed into one of the eight independent **S-boxes**

$$S_1, S_2, \ldots, S_8 : \{0,1\}^6 \to \{0,1\}^4,$$

whose outputs are concatenated into a 32-bit block. Finally,

$$P : \{0,1\}^{32} \to \{0,1\}^{32}$$

is just a fixed permutation of the 32 bits. It is sometimes called the **P-box**.

Before being input to the Feistel network, the plaintext is subject to a fixed initial permutation (IP) of its bits; similarly, the output of the network takes the inverse permutation to become the ciphertext (these permutations are essentially irrelevant to its security). The overall structure of DES is depicted in Fig. 3.1. The details – the expansion permutation $E$, the tables describing the S-boxes, the permutation $P$ and the initial permutation IP, and the **key schedule** which selects 48 bits of the key for each round – can be found in many textbooks and webpages and will not be repeated here. You will need to be familiar with the details for the second practical.

DES was designed for hardware implementation: operations such as permutation of bits is very fast in hardware (cross some wires!) and slower in software. The S-boxes have particularly efficient silicon implementations. The speed of DES in software is a bit disappointing.

DES was carefully designed with all the knowledge of 1970s cryptography and cryptanalysis. The S-boxes are the nonlinear component, essentially fixed tables giving one of 16 outputs to each of 64 possible inputs (everything else in DES is linear). They ensure *confusion*, because any equation relating the input bits to output bits will necessarily be very complicated[10].

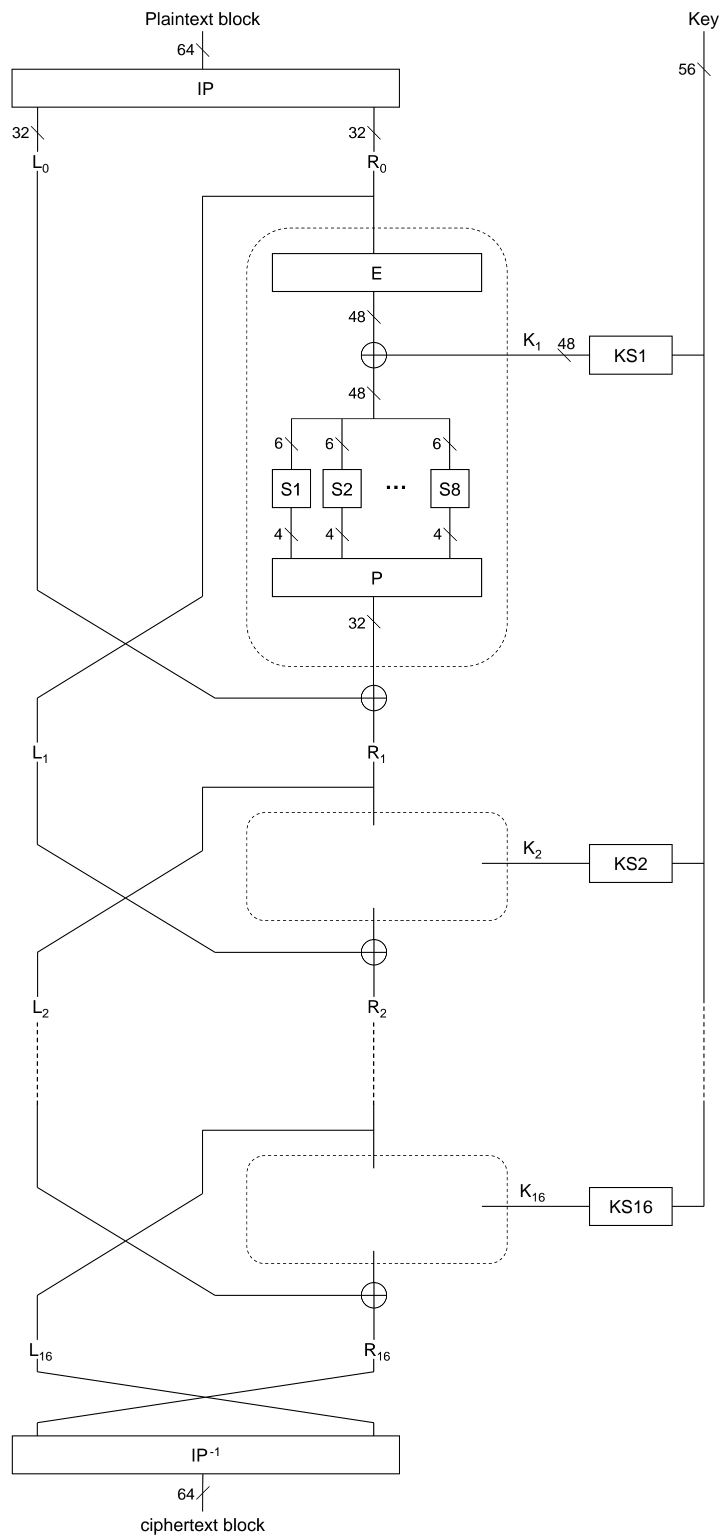
The P-box permutations are primarily responsible for *diffusion*. Suppose that one bit is changed in $R_n$. After the expansion map and XOR with the round key, this means that one or two inputs to the S-boxes are altered, which affects all 4 output bits of each box (any or all of them might be changed, depending on the other bits). Then the P-box moves those changes so that, when the next Feistel round occurs, the affected bits are now dispersed into many S-boxes affecting very many bits, and within very few rounds all output bits are affected. The same is true of changing one bit of the key.

For example, if we write the bitstreams as sequences of hexadecimal nibbles[11], standard

[10] That has not stopped cryptanalysts trying to solve them, as happens in certain attacks on DES.

[11] 4 bits per character.

Figure 3.1: Outline of the structure of DES.



DES gives

$$E_{\texttt{AAAAAAAAAAAAAAAA}}(\texttt{0000000000000000}) = \texttt{7D239B3EF2E153C4};$$

changing one bit of the message gives

$$E_{\texttt{AAAAAAAAAAAAAAAA}}(\texttt{0000000010000000}) = \texttt{6E9483109EB171FA};$$

and changing one bit of the key gives

$$E_{\texttt{AAAAAAAAAAAAAAAB}}(\texttt{0000000000000000}) = \texttt{CF53E71144F80DDF}.$$

The fact that the S-boxes are many-to-one (in fact they all map exactly four inputs to each possible output) is also important. It means that it is impossible to reason backwards, from the output of the Feistel round back up towards the subkey, because the number of possibilities multiplies as you go back through the rounds.

There have been very many attempts to attack DES, none of them very successful. An interesting attack is called **differential cryptanalysis**, which looks at how small changes in the input cause changes in the output which, sometimes, reveal partial information about the key. Differential cryptanalysis was published in 1990, but it turns out that the functions which make up the S-boxes of DES are tuned to be resistant to it. It is undisputed that the NSA designed the S-boxes used for DES, which tells us that the NSA already knew about differential cryptanalysis in the 1970s.

The best attack, involving multiple linear approximations to the DES round function, requires approximately $2^{41}$ known plaintext-ciphertext pairs. It is not actually practical, but the combination of increasing computing power with decreasing costs means that the plain exhaustion attack (which requires only one known plaintext-ciphertext pair!) is now a reasonable option, either with dedicated hardware or with distributed computing.

Indeed, the main weakness of DES is simply that the key size is a bit too small. This was a criticism even when DES was first proposed, and it is now a serious weakness (Triples DES, see section 3.6, is still considered a decent cipher). The reasoning behind the decision to use only 56 bits for the key, when the same structure could easily have accommodated 64 or even more, was always rather vague. It is widely suspected that the NSA put pressure on IBM to keep the key size down, to make it easier for them to eavesdrop on others' secrets...

### 3.5.3 AES

When it became clear that computational power was making DES insecure, an open competition was held to find a replacement. The successor to DES, also published as a US standard (National Institute of Standards and Technology, 2001), is called the **Advanced Encryption Standard** (AES).

AES operates on 128 bit plaintext blocks, and uses a 128-bit key (variations with 192- and 256-bit keys are available). It has components which will be familiar. It is not a Feistel network, but has a similar structure: a 128 bit block is kept as a $4 \times 4$ matrix of bytes, which undergoes ten rounds. In each round some columns are exchanged and rows reordered (permutation), each byte undergoes a fixed nonlinear substitution (S-box), and a subkey is XORed with the block. There is an additional step, where the elements of the matrix are combined together in a complicated bitwise operation which ensures strong diffusion. The details can be found in (Kaufman et al., 2002, §3.5).

AES was finalized in 2001 and is considered to be a secure cipher. Some attacks reduce the security of the 192- and 256-bit versions, and there are attacks on versions of AES with fewer than ten rounds. Some specific flawed implementations can be attacked, but at the time of writing there were no attacks on the full 128-bit AES algorithm any better than the completely-impractical exhaustion attack.

## 3.6 Repeated Encryption

Suppose we are given a cipher with the same plaintext space as ciphertext space $\mathcal{C} = \mathcal{M}$ (as usual), which we think might have slightly too-small a keyspace $\mathcal{K}$,

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}; \quad D : \mathcal{K} \times \mathcal{M} \to \mathcal{M}.$$

An example might be DES, which seems to be quite a good cipher except for its 56-bit keyspace.

We might try to strengthen it against an exhaustion attack by encrypting twice. That is, we create a new cipher

$$E' : \mathcal{K} \times \mathcal{M} \to \mathcal{M}; \quad D : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$$

$$E'_k(m) = E_k(E_k(m)); \quad D'_k(c) = D_k(D_k(c)).$$

Hopefully you can see that this provides little extra security against an exhaustion attack. Read in (Kaufman et al., 2002, 4.4.1.1) if you need a hint.

So we might try encrypting twice under two different keys. That is, we create a new cipher

$$E'' : \mathcal{K}' \times \mathcal{M} \to \mathcal{M}; \quad D'' : \mathcal{K}' \times \mathcal{M} \to \mathcal{M}$$

with $\mathcal{K}' = \mathcal{K} \times \mathcal{K}$ defined by

$$E''_{\langle k,k' \rangle}(m) = E_{k'}(E_k(m)); \quad D''_{\langle k,k' \rangle}(c) = D_k(D_{k'}(c)).$$

This new keyspace has cardinality the square of the old keyspace, or twice as many bits. For example, "double DES" would have $2^{112}$ possible keys. Apparently, an exhaustion attack on $E''$ would square the number of steps in the exhaustion attack on $E$. However, this is not the case.

The attack known as **meet-in-the-middle**[12] trades space for time. This is a KPA.

**Meet-in-the-middle attack**

(1) Obtain a plaintext $m$ with known encryption $c$.
(2) For each $k \in \mathcal{K}$, compute $E_k(m)$ and store $\langle k, E_k(m) \rangle$ in list $L$.
(3) For each $k' \in \mathcal{K}$, compute $D_{k'}(c)$ and search for $\langle k, D_{k'}(c) \rangle$ in $L$; if found, $\langle k, k' \rangle$ is called a *candidate pair*. Store a list of all candidate pairs.
(4) Test the candidate pairs $\langle k, k' \rangle$ against a new plaintext $m'$ with known encryption $c'$. Discard the pair if $E_{k'}(E_k(m')) \neq c'$.
(5) Repeat the previous step until only one candidate pair remains: this is the double encryption key.

For step (3) to be efficient, $L$ should be sorted by second component after step (2), or stored in a suitable data structure.

The attack works because $E_k(m) = D_{k'}(c)$ if $E_{k'}(E_k(m)) = c$. It requires $O(2^{n+1})$ en/decryptions for steps (2)-(3) if $\mathcal{K}$ has $n$ bits, and the rest is insignificant. That is to say, double encryption only requires approximately double the number of steps to attack as does single encryption, rather than a square number.

[12] under no circumstances to be confused with **man-in-the-middle**, see chapter 7.

Meet-in-the-middle is a nice trick, though it requires a lot of working space (as you will show on the first problem sheet). A popular way to improve the security of a cipher with limited keyspace is to use a form of *triple* encryption.

$$E'''_{\langle k,k',k''\rangle}(m) = E_{k''}(D_{k'}(E_k(m)))$$

with the obvious corresponding decryption. The use of $D_{k'}(-)$ in the middle does defend against some mild attacks, and also makes the triple encryption backwards-compatible with single encryption if $k = k' = k''$. Triple encryption is used to strengthen DES.

Of course, one cannot improve the security of a completely flawed cipher, like the monoalphabetic substitution cipher, by repeating it. Indeed, since permutation operations form a **group**, multiple encryptions are equivalent to just one encryption under a combined key. It has been proved that DES encryption is not a group, so multiple encryption should improve security.

The FIPS standard which defines DES allows Triple DES with arbitrary $k, k', k''$, which gives 168 bits of keyspace, but a meet-in-the-middle attack can reduce the exhaustion attack from $2^{168}$ to $2^{112}$ steps. It has become common to set $k = k''$, thus there are 112 bits of keyspace involved. There are recent KPA or CPA attacks which exploit the triple encryption structure and take about $2^{80}$ steps. This is good security for now, but perhaps not for secrets which must remain so for many years, particularly if we take into account the possibility that more efficient attacks may appear in future.

## 3.7 Encrypting Longer Messages

We have not said how to encrypt messages longer than a single plaintext. In section 3.3 we effectively used single-character encryption and a sequence of plaintexts encrypted to the corresponding sequence of ciphertexts, but this is not secure: it is what leads to the frequency analysis attack. There are several better possibilities which appear in the official standard with DES, and we cover them in this section.

This material can be found in many places online and is very well covered in chapter 4 of the course text (Kaufman et al., 2002), so these lecture notes only include outline material.

### Electronic Codebook (ECB)

Each plaintext block is encrypted separately:

$$c_i = E_k(m_i).$$

Decryption is similar. It leads to weaknesses if plaintexts might include repeated or similar blocks (highly plausible).

### Cipherblock Chaining (CBC)

The encryption of block $i$ is XOR-ed with the plaintext of block $i + 1$ before encryption, acting as a pseudorandom "blinding". The sequence needs to be started with an **initialization vector** $IV$.

$$\begin{aligned} c_1 &= IV, \\ c_{i+1} &= E_k(m_i \oplus c_i). \end{aligned}$$

Note that the $IV$ must be transmitted to Bob, who will discard it after use:

$$m_i = c_i \oplus D_k(c_{i+1}).$$

The initialization vector is important, and it is risky to leave it constant: if the same IV is used every time, the attacker will be able to tell when identical plaintexts have been sent. It is better to randomise it, and change the encryption key often enough that IVs are not repeated. Introducing a random component also defeats a chosen plaintext attack and we will see the importance of randomised encryption in chapter 5.

### Output Feedback Mode (OFB)

Output feedback mode simulates the one-time pad, using an iterated encryption to generate a pseudorandom stream $\langle r_2, r_3, \ldots \rangle$ which can simply be XORed with the plaintext. In fact, it is effectively a stream cipher (see section 3.8) and so is good for low latency situations.

$$\begin{aligned} c_1 = r_1 &= IV, \\ r_{i+1} &= E_k(r_i), \\ c_{i+1} &= m_i \oplus r_{i+1}. \end{aligned}$$

The pseudorandom stream can be generated on demand or in advance. As with CBC, the IV is important and must be transmitted to the recipient. A weakness arises if the pseudorandom sequence ever repeats a block: it will then get into a cycle. It is important to change the key often enough to make this unlikely. An alternative is the following.

### Counter Mode (CTR)

Counter mode creates pseudorandom blocks by encrypting an increasing sequence:

$$\begin{aligned} r_i &= E_k(IV \oplus i) \\ c_1 &= IV \\ c_{i+1} &= m_i \oplus r_{i+1} \end{aligned}$$

CTR mode will not get into a cycle. It has some theoretical guarantees of security under certain assumptions about the block cipher used, and unlike CBC or OFB it can be parallelized because the cipher blocks do not depend on each other. However, it seems to be little used.

### Cipher Feedback Mode (CFB)

There is also a mode known as Cipher Feedback Mode, which is a modification of OFB which removes some of the advantage of a stream cipher but adds a property known as **self-synchronization**. We will not cover it in this course, but see (Kaufman et al., 2002, §4.2.4) for details.

We finish with a warning. Some authors discuss the **robustness** of these block modes, which means that small changes made to the ciphertext cause only small changes in the decrypted plaintext (in the case of ECB only the block changed is affected, in CBC the following block is also affected, in OFB/CTR only the bits changed are affected, and things are more complex for CFB). Robustness might be seen as an advantage, if the ciphertext is to be transmitted through a noisy channel such as radio, or a weakness if the recipient wants to be able to detect tampering of the ciphertext. The latter would be considered in the context of an attacker who can tamper with the ciphertext, perhaps to cause a desired alteration in the plaintext, but it is a mistake to think that symmetric cryptography alone, no matter what block mode is used, guarantees the **integrity** of the message. Chapters 4–6 describe the tools used for integrity of a transmitted message.

## 3.8 Stream Ciphers

Stream ciphers attempt to simulate the perfectly secure one-time pad, using a cryptographic key to generate (or **seed**) a pseudorandom stream of bits, or blocks of bits, which is then XORed with the plaintext for encryption, and XORed again for decryption. A block cipher in OFB or CTR mode is an example of this, but the pseudorandom stream

could be generated by other means and need not use a block cipher at its heart. Stream ciphers do not provide perfect security because the pseudorandom stream is not truly random: the symbols are not (quite) independent, and even very weak dependencies can often lead to powerful exploits.

In fact, it is probably fair to say that stream ciphers are generally less secure than block ciphers. But they have a particular advantage in low latency applications. It could be tiresome to wait for a complete block before encryption and transmission, or expensive in bandwidth to pad mostly-empty blocks more often, but a stream cipher can produce pseudorandom bits and send the cipher bits immediately (or, like OFB/CTR, produce blocks of bits in advance so that they can be used on demand).

The whole development of stream ciphers is different from that of block ciphers, being closer to that of pseudorandom number generation than true cryptography, and the terminology is a bit different. The construction of most stream ciphers involves a hidden **state** from a set of possible states $\mathcal{S}$, an **initialization function** $I : \mathcal{K} \to \mathcal{S}$ which constructs an initial state from the key, an **update function** $U : \mathcal{S} \to \mathcal{S}$ which alters the state, and an **output function** $f : \mathcal{S} \to \Sigma$, where $\Sigma$ is the alphabet for the stream cipher (usually bits or bytes, for low latency applications). Every time the output function is called to request a new symbol from the stream, the state is updated.

The most widely-used stream cipher is RC4, which features in the SSL protocol and WEP wireless security. It has some weaknesses but is *extremely* fast and the weaknesses can be avoided in careful use. (The most important thing is to discard the first few bytes of the output stream.)

Although we will not analyse its security, we can give a brief description of RC4. Let $B = \{0, 1, \ldots, 255\}$ be the set of unsigned byte integers, and $S_B$ the set of all permutations of $B$. Then

$$\mathcal{K} = B^{256} \text{ and } \mathcal{S} = S_B \times B \times B$$

i.e. a key is a sequence of 256 bytes (shorter keys are okay too: they are repeated cyclically to make them up to length 256) and the state is a tuple $\langle \pi, x, y \rangle$ where $\pi$ is some permutation of $B$.

The initialisation is $I(K) = \langle \pi, 0, 0 \rangle$ where $\pi$ is the permutation given by the program

**for** $i = 0 \mathinner{..} 255 :\ \pi(i) = i$

$k = 0$

**for** $i = 0 \mathinner{..} 255 :\ k = k + K_i + \pi(i);$ **swap** $\pi(i), \pi(k)$

The update function is $U\langle \pi, x, y\rangle = \langle \pi', x', y'\rangle$ where

$$\begin{aligned} x' &= x + 1 \\ y' &= y + \pi(x) \\ \pi' &= \pi \text{ swapping } \pi(x'), \pi(y') \end{aligned}$$

and the output function is

$$f\langle \pi, x, y\rangle = \pi(\pi(x) + \pi(y))$$

(in all of the above the arithmetic is implicitly mod 256).

The weaknesses found so far in RC4 (see (Fluhrer et al., 2001) for some of the first serious attacks) are not so great as to outweigh its extreme ease of use, which is why it continues to be deployed today. Although there are weaknesses in WEP which exploit the correlation of the first RC4 output bytes with its key, they are primarily flaws in the way WEP uses RC4, and indeed there are other flaws in WEP not related to RC4 (notably its very short IV).

## 3.9 Concluding Remarks

We have discussed the formal definition of ciphers, seen some weak examples, and explained why perfect security is not usually practicable. Then we examined the development of modern ciphers. Given that symmetric key cryptography is a maturing field, and that experts consider AES to be secure, have we solved the security problem of confidentiality in communications and data storage?

Not really. All we have done is reduce the problem of confidentiality of the message to the problem of sharing a secret cipher key (and this had better be done confidentially, with integrity and authentication too). Thankfully, we will address these problems later in the course. We may also note that only the *contents* of the message are made confidential, and often at least some information about the *length* is revealed (even with perfectly secure cryptography). Cryptography certainly reveals the fact that the confidential information exists...

We have outlined the design and construction of some commonly-used ciphers. But this is *not* meant to encourage you to design or implement a cipher yourself: it is all too easy

to make mistakes which introduce weaknesses, and much better to rely on the community of cryptographers to invent, refine, and create safe implementations of good ciphers.

We conclude by stressing that symmetric key cryptography is designed to offer confidentiality, not integrity, authentication, or non-repudiation. It is easy to see why conditions (i)-(iii) in subsection 1.3.4 are not satisfied by the simple use of symmetric key ciphers. In particular, there is no defence against replay or reordering of the encrypted messages.

## Bibliography

Fluhrer, S., Mantin, I., & Shamir, A. (2001). Weaknesses in the key scheduling algorithm of RC4. In *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science* (pp. 1–24). Springer. Available at `http://www.crypto.com/papers/others/rc4_ksaproc.pdf`.

National Bureau of Standards (1993). Data encryption standard. Federal Information Processing Standards Publication 46-3, US Department of Commerce. Available at `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

National Institute of Standards and Technology (2001). Advanced encryption standard. Federal Information Processing Standards Publication 197, US Department of Commerce. Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4), 656–715. Available at `http://www.alcatel-lucent.com/bstj/vol28-1949/articles/bstj28-4-656.pdf`.

# Chapter 4

# Cryptographic Hashes

---

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §2.6, 5.1-5.2, 5.4-5.6, 9.1, chapter 10 |
| **Alternatives & further depth:** | Gollmann §14.3, chapter 4 (very sketchy)<br>Ferguson et al. chapter 5, §21.2<br>Menezes et al. §9.1-9.4, 9.6-9.7, 10.2 |

---

Broadly speaking, a cryptographic hash function condenses an arbitrary-length piece of data into a short, fixed-size, block called a **digest** or **hash**. The idea is that the digest can "represent" the data in situations when a fixed-length sequence of bits is required, and without disclosing the data itself. There are many such situations, and in this chapter we will consider three particular applications:

(i) In a multi-user system where users log in by password, the password database itself presents a potential vulnerability. If the hashes of passwords are stored in the database, and the log in program checks the hash of the supplied password against the database entry, disclosure of the database does not immediately expose all users' passwords. But extra steps must be taken to avoid offline dictionary attacks.

(ii) In symmetric encryption, Alice and Bob need to share a secret key dozens of bits long, which would be very difficult to memorise. If they instead share a memorable password, they can generate the secret key by hashing the password. Further steps must be taken to avoid dictionary attacks.

(iii) If a message is transmitted alongside its own hash, tampering with the message will break the relationship with the hash. Used with symmetric key encryption, integrity of the message can be assured.

There are other uses too, most importantly the **digital signatures** which we will see in chapter 6 but also the general technique of **commitment**, which is to demonstrate that you know a certain piece of information without revealing the information until later.

These applications do not require exactly the same conditions of the hash function used, but the conditions tend to be met by similar constructions and common practice is to find a function satisfying all the conditions and use it for all the applications.

The hash functions in this chapter will be **unkeyed**: their only parameter is the data they digest. In chapter 6 we will consider keyed hash functions, which provide greater security.

## 4.1 One-Way Functions and Cryptographic Hashes

The word "hash" appears in many areas of computer science, and we stress that the hash functions used in some algorithms (e.g. hash tables) have nothing to do with those used in cryptography. We should also point out that the all the terminology relating to cryptographic hash functions is very confused, with different authors using completely different terms for each concept, so one must read carefully in this area.

Suppose that $h : D \to \{0,1\}^n$, where $D$ is the domain and $n$ is some fixed output length. Here are some possible conditions we might put on $h$:

**(1A)** $D = \{0,1\}^n$.

**(1B)** $D = \{0,1\}^{n+k}$ with $k > 0$.

**(1C)** $D = \{0,1\}^*$. This condition is called **compression**.

**(2)** For all $x \in D$, $h(x)$ is easy to compute.

**(3)** For almost all $y \in \mathrm{Im}(h)$[1], it is computationally infeasible to find an $x$ such that $h(x) = y$. This condition is called **preimage resistance** or **one-way**.

**(4)** Given $x \in D$, it is always computationally infeasible to find $x' \neq x$ such that $h(x) = h(x')$. This condition is called **2nd-preimage resistance** or **weak collision resistance**.

**(5)** It is computationally infeasible to find any pair $x \neq x'$ such that $h(x) = h(x')$. This condition is called **collision resistance** or **strong collision resistance**.

The book (Menezes et al., 1996, §9.2) gives certain collections of these properties particular names:

A **one-way function** satisfies **(1A)**, **(2)**, and **(3)**.

A **compression function** satisfies **(1B)**, **(2)**, **(3)**, **(4)**, and **(5)**.

A **one-way hash function** satisfies **(1C)**, **(2)**, **(3)**, and **(4)**.

A **cryptographic hash function** satisfies **(1C)**, **(2)**, **(4)**, and **(5)**.

We repeat that different authors use the same terminology in different ways and the terms "one-way" and "hash function" are often abused by vague usage. It is probably true to say that almost everyone agrees on the terminology **cryptographic hash function** as above, except that some might include **(3)** as well. Thankfully, **(3)** almost always follows from **(4)**, though there are some artificial counterexamples. It is typical of this messy area, and unfortunate, that the property of "compression" is not satisfied by a "compression function"...

Condition **(3)** is carefully worded. For most $y$ there will probably be infinitely many different $x$ which map to $y$, but we only need one such $x$ to make a preimage. On the other hand, if we pick an output $y$ that does not correspond to *any* input $x$ then of course it would not be possible to find $x$ such that $h(x) = y$. More importantly, if we were to pick a random input and compute $h(x) = y$, it then becomes very easy to find a preimage for $y$! That is why the condition requires it be difficult to find preimages for *almost all* possible outputs. One can always perform a **partial exhaustion attack** by tabulating

[1] $\mathrm{Im}(h)$ denotes the **image** of $h$, the set $\{h(x) \mid x \in D\}$.

some hash values and then hoping that the requested preimage has already been found. A more precise, complexity-style definition of (3) would say that it is infeasible to find inputs corresponding to a non-negligible proportion of possible outputs.

For condition **(5)**, note that the compression property implies that the function must be many-to-one, because the domain is infinite and the codomain finite, so infinitely many collisions exist. But it should be computationally infeasible to find any of them[2].

In practice, the one-way and collision-resistance conditions require the hash function to have similar diffusion properties (changing one bit of the input changes potentially all bits of the output, unpredictably) as do symmetric key ciphers. However, the community has had less success in designing good hash functions than good encryption.

### 4.1.1 Proofs About Hash Function Properties

We can prove results about these conditions, for example transferring collision resistance from a function to something based on it. We almost always prove the *contrapositive*, because it is much easier to assume that something is *not* preimage, 2nd-preimage, or collision resistant, and then to prove that something else cannot have the property either. It is worth writing explicitly:

- **(¬3)** $h$ is **not** preimage resistant if, for *more than a negligible proportion of* $y \in \mathrm{Im}(h)$, it is computationally feasible to find an $x$ such that $h(x) = y$.
- **(¬4)** $h$ is **not** 2nd-preimage resistant if, from *even one* $x \in D$, it is computationally feasible to find $x' \neq x$ such that $h(x) = h(x')$.
- **(¬5)** $h$ is **not** collision resistant if it is computationally feasible to find *even one* pair $x \neq x'$ such that $h(x) = h(x')$.

Remember that it only requires *a few* invertible outputs to falsify preimage resistance, a 2nd-preimage of *one* output to falsify 2nd-preimage resistance, and the generation of *a single* pair of colliding inputs to falsify collision resistance.

An example of a proof about collision resistance is

[2] As with symmetric key encryption, one can formalise the definition of computational infeasibility for cryptographic hash properties. For example, the one-way property could be defined as: given randomly chosen $Y$, for any function $A(Y)$ computable in polynomial time the difference in probability that $A$ finds a preimage, compared to random guessing, $|\mathrm{P}[h(A(Y)) = Y] - 2^{-n}| \leq \nu(n)$, where $\nu(n)$ is a negligible function.

**Claim 4.1** Let $f, g : \{0,1\}^* \to \{0,1\}^n$. Define the function $h(x) = f(x) \parallel g(x)$. Then if *either* $f$ *or* $g$ is collision resistant then so is $h$.

**Proof** A collision for $h$ means two inputs $x$ and $x'$ such that $h(x) = h(x')$, which in turn implies $f(x) = f(x')$ and $g(x) = g(x')$. Therefore, if it is easy to find a collision for $h$ then it is easy to find a collision for $f$ and a collision for $g$. The contrapositive of this statement is the one required. ■

You will see more examples in the exercises.

### 4.1.2 From Compression Functions to Hashes

A common use of compression functions is to construct, by iteration, one-way hash functions and cryptographic hash functions.

Suppose we have a function $g : \{0,1\}^{n+k} \to \{0,1\}^n$. Let us split the input to a compression function into two parts, so that it becomes

$$g : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^n.$$

Then we can take an arbitrary-length input $x$, split it into a sequence of $k$-bit blocks $\langle x_1, \ldots, x_m \rangle$, and then perform an iterative process

$$h\langle\rangle = IV$$
$$h\langle x_1, x_2, \ldots, x_m \rangle = g(h\langle x_1, \ldots, x_{m-1} \rangle, x_m).$$

where $IV$ is some initialization vector.

This is the most common way to create one-way and cryptographic hash functions, but it is very easy to make them insecure. For one thing, we will need to decide how to pad incomplete blocks and it is *vital* to get this right otherwise it creates easy-to-find collisions, as we will see in section 4.6. If done correctly, however, it is possible to transfer preimage resistance, 2nd-preimage resistance, and collision resistance properties from $g$ to $h$.

For now, let us look at some quick examples of insecure hashes. The simplest would be to use the above construction with $n = k = 32$ and the function $g(x, y) = x + y \pmod{2^{32}}$ where $x$ and $y$ are 32-bit unsigned integers. If the IV is zero, this is the

**checksum** hash which treats the input as 32-bit words and computes their sum. It does have the compression property, but it is not preimage resistant (since every 32-bit input maps to itself), nor 2nd-preimage resistance (since appending an extra 32-bit block of zeros does not change the output), nor collision resistant (for the same reason). Many other common checksums such as **CRC** are also not preimage, 2nd-preimage, or collision resistant because they have linearity properties.

It might seem natural to use a block cipher as a compression function, but there can be subtle flaws. Let $E_k(x)$ denote the encryption of a block cipher using some fixed (known) key $k$. The iteration

$$\begin{aligned} h\langle\rangle &= 0 \text{ (a block of zeros)} \\ h\langle x_1, \ldots, x_m\rangle &= E_k(h\langle x_1, \ldots, x_{m-1}\rangle \oplus x_m) \end{aligned}$$

is not collision resistant, as we can show without attacking the encryption component itself. Pick any $x$ such that $x$ is exactly two blocks long, $x = x_1 \,\|\, x_2$. Set $x' = x'' \,\|\, (E_k(x'') \oplus x'')$ where $x'' = E_k(x_1) \oplus x_2$. Then compute

$$\begin{aligned} h(x) &= E_k(E_k(x_1) \oplus x_2), \\ h(x') &= E_k\big(E_k(x'') \oplus (E_k(x'') \oplus x'')\big) = E_k(x'') = E_k(E_k(x_1) \oplus x_2), \end{aligned}$$

and $x \neq x'$ almost certainly.

In one of the problem sheets you will find additional attacks on (a more general version of) this function, including its preimage resistance. When the function is keyed by $k$ this is called a CBC-MAC – see chapter 6 – but hash functions are **unkeyed** and $k$ must be a publicly known component, which is the source of its weakness.

For the rest of this chapter, until section 4.6, we will assume that perfect one-way and cryptographic hash functions exist and turn to the applications.

## 4.2 Exhaustion Attacks on Hashes

An important distinction between one-way and cryptographic hash functions is the difficulty of a brute-force exhaustion attack. Brute forcing a preimage $x$ such that $h(x) = y$, or a second preimage $x'$ such that $h(x) = h(x')$, means testing all members of the domain one-by-one until a solution is found. For one-way and cryptographic hash functions the

domain is the infinite set $\{0,1\}^*$! But in practice a solution is almost always to be found in an input sequence of length $n$ (the same size as the output). The average time to find a solution is then $2^{n-1}$, similarly to the exhaustion attack for symmetric key ciphers. A good one-way hash function, then, should offer "$n$ bits of security" in the sense that it takes about half of $2^n$ tries to break the one-wayness.

Collision resistance is different. An exhaustion attack needs to find $x$ and $x'$ such that $h(x) = h(x')$, otherwise unconstrained. The attack should proceed by computing hashes of random inputs, recording the outputs in a suitable data structure, until a duplicate output is found. This takes much less time than brute forcing a preimage, as we prove next.

### 4.2.1 The Birthday Paradox

The birthday "paradox" is commonly stated thus: you only need 23 people in a room for there to be at least a 50% chance that two share the same birthday[3]. This is a "paradox" because the number 23 seems is much smaller, in relation to the possible number of birthdays 365, than mathematically ignorant people expect. The reason is that, for no coincident birthdays to occur, the first person in the room is unconstrained, the second person must dodge one date, the third must dodge two dates, the fourth three dates, and so on. As more people enter the room it gets less likely that the next person does not duplicate someone else's birthday.

This is of direct relevance to the exhaustion attack on hashes: for the exhaustion attack to fail to complete by step $t$, all randomly-chosen inputs up to and including step $t$ must have produced a fresh hash output. Let us write $h = 2^n$ for the number of possible hash outputs, and say that $T$ is the random variable representing the step at which we find the first duplicate hash output, the first collision. If the inputs are randomly chosen and the hash produces uniformly random outputs[4] then we can write

$$\mathrm{P}[T > t] = 1 \cdot \left(1 - \frac{1}{h}\right) \cdot \left(1 - \frac{2}{h}\right) \cdots \left(1 - \frac{t-1}{h}\right).$$

This formula is difficult to work with, but there is a good approximation which comes from the Taylor expansion $\exp(x) = 1 + x + O(x^2)$, which implies that $1 - x \approx \exp(-x)$

[3] Under the assumption that birthdays are uniformly distributed over 365 days of the year, and ignoring leap years.

[4] If the outputs are not uniform then collisions become even more likely, and the running time calculations will be overestimates.

for small $x$. It gives

$$\begin{aligned}
\mathrm{P}[T > t] &\approx \exp\left(-\frac{1}{h}\right)\exp\left(-\frac{2}{h}\right)\cdots\exp\left(-\frac{t-1}{h}\right) \\
&= \exp\left(-\frac{1}{h}\sum_{i=1}^{t-1} i\right) \\
&= \exp\left(-\frac{t(t-1)}{2h}\right) \\
&\approx \exp\left(-\frac{t^2}{2h}\right)
\end{aligned}$$

With these simplifications we are in position to compute, approximately, the number of steps at which the probability of finding two equal hashes has risen to 50%. We solve $\mathrm{P}[T > t] = 0.5$,

$$\exp\left(-\frac{t^2}{2h}\right) \approx \frac{1}{2}$$

which gives

$$t \approx \sqrt{2\ln(2)h}.$$

($h = 365$ gives the standard birthday paradox.) Note that this is just a constant (close to 1) times $h^{0.5}$.

Or perhaps we care about the *average* number of steps to the first collision, since we will not stop until we find one. This is more difficult. First, recall that for non-negative integer-valued random variables the mean has a nice alternative formula

$$\mathrm{E}[T] = \sum_{t=0}^{\infty} t\,\mathrm{P}[T = t] = \sum_{t=0}^{\infty} \mathrm{P}[T > t]$$

(because the second sum counts $P[T = 0]$ zero times, $P[T = 1]$ one time, and so on). So

$$\mathrm{E}[T] = \sum_{t=0}^{\infty} \mathrm{P}[T > t] \approx \sum_{t=0}^{\infty} \exp\left(-\frac{t^2}{2h}\right)$$

Then we need a piece of really cheeky mathematics to approximate an impossible infinite sum by the integral of the area under a curve, and then recognise the integral as one

relating to the Normal distribution (if you have not seen this, you may take it as read):

$$\mathrm{E}[T] \approx \int_0^\infty \exp\left(-\frac{t^2}{2h}\right) \mathrm{dt} = \sqrt{\frac{\pi}{2}h}.$$

Once again, this is a constant (close to 1) times $h^{0.5}$.

Mathematicians might be horrified about the loose use of approximations here. Rest assured that the analysis has been done properly (by Ramanujan, no less!) and the precise result is

$$\mathrm{E}[T] \sim \sqrt{\frac{\pi}{2}h} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2h}} + O(1/h).$$

### 4.2.2 Consequences of the Birthday Paradox

The birthday paradox means that we only have to do $O(2^{n/2})$ computations of an $n$-bit hash, until we find a value we have seen before. Thus the exhaustion attack on the collision resistance of cryptographic hash functions is much cheaper than on preimage resistance. It means that, when we care about collisions, we should choose hash functions with output about twice as long as the corresponding key in symmetric cryptography. For this reason, even basic hash functions have 128-bit outputs.

Alert readers will have wondered about the practicalities of the exhaustion attack: remembering every hash value we have seen before may take a lot of space, and searching through those values might take a non-negligible amount of time. Instead we should use methods which iterate the hash function until a **cycle** results, at which point a collision has been found. There are a number of linear-time, constant-space algorithms for cycle finding, e.g. Floyd's "tortoise and hare" algorithm (Knuth, 1998, Ex 3.1.6).

## 4.3 Hashes and Passwords

One of the most basic mechanisms in computer security is the password. At some point in time the user $A$ is initialised by creating a **username** – some unique identifier – and an initial password. The pair

$$\langle \text{Username}_A, \text{Password}_A \rangle$$

will be held in some database by the host and there will probably be some process by which the password can or must be changed from time to time. When necessary, the user enters their username and password to authenticate themselves to the host (note that this separates **identification** of the user, which is achieved by the username, from the **authentication** of that identity by knowledge of the password).

Modern computer systems prevent a brute force attack on a user's password by limiting the number of incorrect guesses which can be made, either by refusing any more authentication by the user (requiring superuser action to renew the password) after too many attempts, or simply by replying slowly to incorrect guesses.

However, the system must store the password database somewhere. In a sensible system the password database would be readable only by privileged users such as the superuser and the login process, but this security might be bypassed[5]. Furthermore, users have bad habits and choose the same passwords for different systems. If we want security in depth, we should hope that disclosure of the password database does not necessarily breach the security of every user of the system, or risk the security of other systems.

One could encrypt the password database. But the decryption key must exist somewhere on the computer system too, so this does not provide significant additional security. (Many e-commerce websites provide a **forgotten password** feature which will send a user their password via email. This is subject to many security risks in itself, but also indicates that the plaintext password is available to the host and therefore potentially disclosable. It also highlights the risk of using the same password on multiple hosts.)

A good, simple, solution is for the host not to store the user passwords at all. Instead, they store the results of a one-way hash function on the passwords: the password database contains

$$\langle \textsc{Username}_A, h(\textsc{Password}_A) \rangle,$$

and the login process checks whether the supplied password computes to the correct hash. This is the technique used by UNIX systems, which historically allowed the password database to be readable by any user. Unfortunately the hash used was not wonderfully

[5]For example, a logged-in superuser might be away from their keyboard, and an attacker uses the window of opportunity to copy the password database to a file with universal read permissions. This exploit once allowed the lecturer to break into the computer system at his school. Please note: this action predated the Computer Misuse Act 1990. It is unethical, and now illegal, to obtain unauthorized access to a computer resource.

secure: modern versions use a different hash and keep the password hashes secret just in case.

We do not need the function $h$ to have all the properties of a cryptographic hash. We do not even need it to accept arbitrary-length inputs, as long as it accepts inputs long enough for secure passwords. Collisions do not present a security risk, and neither do 2nd-preimage attacks. The only requirements are **(2)** that it be easy to compute and **(3)** one way: preimage attacks are serious, because they allow an attacker who has obtained the password database to deduce the users' passwords and then, presumably, impersonate them. In practice, however, one tends to use an algorithm designed as a full cryptographic hash, because they are easily available and the extra properties should not hurt.

### 4.3.1 Offline Attacks

Let us examine the situation where an attacker has obtained the password database which contains usernames and (securely one-way) hashed passwords. Now they are in a position to mount an **offline attack**, meaning that they do not need to interact with the host. A brute-force exhaustion attack can test every possible sequence of characters to see whether it maps to one of the hashed passwords, and so one possible password can be tried in the time it takes to compute one hash. (As with cryptography, we must assume that the enemy knows the system, i.e. the hash function used.)

It is for this reason that many systems require **strong passwords**, which are 1) not too short, 2) not composed of restricted sets of characters such as lowercase-only letters, and 3) not derived from dictionary words. For example, a 6-character password drawn from all 95 printable ASCII characters[6] has $95^6 \approx 2^{39}$ possibilities, which is attackable by brute force given enough determination. And if the characters are restricted to, say, lowercase letters and numbers then we have only $36^6 \approx 2^{31}$ possibilities, much too weak. Additional characters create, of course, exponentially more work for an attacker but even 8-character passwords are insecure if they are drawn from lowercase letters and numbers only. Dictionary words are even more attackable: there are probably fewer than a million ($2^{20}$) English words. One can find password lists including English words, modifications by adding digits or replacing digits by numbers, common pairs of words and proper names, with tens of millions of entries. If the user's password is drawn from one of these lists, it will not take too long for the attacker to find it.

Attackers would like to do their work only once, creating a database of hash values for every combination of printable character passwords up to as big a length as possible. When

[6] 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

new hashes are disclosed, they can quickly be checked against a sorted list of hashed passwords. But even if this has practical computation time, the space requirements are infeasible for longer passwords: to store 128-bit hashes of all $95^6$ 6-character printable passwords would require over 10TB and 7- or 8-character passwords are completely beyond reach. However, there is a time-space tradeoff which allows slower checking of hashes in exchange for much smaller stored data.

The simplest tradeoff is to use a chain of hashes, an idea described by Martin Hellman. We sketch it here, and more details can be found in the original paper (Hellman, 1980). Given a set of possible passwords $P$ we have the hash function $h : P \to H$, where $H$ is the set of possible hashes; to make this tradeoff we need to create a function $g : H \to P$.

Then we pick a large selection from $P$, $\{p_1, \ldots, p_n\}$. For each $p_i$ we compute an iterated chain

$$p_i,\ g(h(p_i)),\ g(h(g(h(p_i)))),\ g(h(g(h(g(h(p_i))))))\ldots$$

for $k$ steps. We store only the first password $p_i$ and the last password which we will call $q_i$ in each chain. We vary $n$ and $k$ so that the storage requirement is manageable, and that we hope to have passed through the hashes of most passwords in $P$.

Then if we are given a hash $y$ to invert, we apply $g$ and check whether this appears as $q_i$ for any $i$. If not, we apply $h$ and then $g$ and try again, repeating up to $k$ times. If at any point we hit $q_i$ we go back to the starting point of the chain $p_i$ and iterate through until we hit $y$ somewhere along the chain.

There are some difficulties with this technique when the chains collide, and an improved version of the technique is known as the **rainbow table**. This uses a family of different functions $g_j : H \to P$, applying $g_j$ at position $j$ of the chain. It makes chains less likely to get tangled.

Various projects are underway to compute rainbow tables for popular hash functions. For example they are available for the NTLM hash (used in modern versions of Windows) for 7-character passwords of any type and 8-character passwords without punctuation. They even come with user-friendly brute forcing software[7].

One defence against precomputed tables is to add a **salt** to the password. This simply means that the password database stores

$$\langle \textsc{Username}_A, h(\textsc{Salt} \,\|\, \textsc{Password}_A) \rangle$$

[7] `http://ophcrack.sourceforge.net/`

where $\textsc{Salt}$ is a fixed string. This effectively pads out the passwords to make them too long for general tables. Even better is to make the salt depend on the username, storing

$$\langle \textsc{Username}_A, h(g(\textsc{Username}_A) \parallel \textsc{Password}_A)\rangle$$

for some function $g$, perhaps itself a hash function. This means that an attack table cannot be built against your own computer system[8].

There are other ways to strengthen hashed passwords against brute force attack. One is sometimes called **hash stretching**, in which the hash is iterated a reasonably large number $n$ times (say $n = 1000$). This makes it $n$ times slower to check the authentication, but also $n$ times slower for an attacker to compute hashes themselves. Another is to compute a randomly salted hash $h(\textsc{Salt} \parallel \textsc{Password}_A)$, where the salt is a small number of random bytes (1–2), and then *discard* the salt. The host will have to check whether *any* possible salt could have lead to the stored hash, and likewise the attacker. Both of these techniques illustrate that condition **(2)** of cryptographic hash functions, that they be quick to compute, is also an advantage to a brute-force attacker.

These attacks are of contemporary importance because the disclosure of (partial) databases of usernames and hashed passwords is quite common at the moment. They are probably obtained by privilege escalation due to insecurities in server scripts which have been written in computer languages with poorly-defined semantics.

### 4.3.2 Password Eavesdroppers and Lamport's Scheme

The normal "login" password protocol could be written like this:

1. $A \to H$ : $\textsc{Username}_A$
2. $H \to A$ : `Password?`
3. $A \to H$ : $\textsc{Password}_A$
4. $H$ grants access if $\langle \textsc{Username}_A, h(\textsc{Password}_A)\rangle$ is in its password database.

This is the protocol notation we will use in chapter 7, but here its meaning should be evident. This protocol is all very well as long as Alice's communication with the host

[8] A contemporary example of the vulnerability of unsalted hashes is described at `http://nakedsecurity.sophos.com/2012/06/06/linkedin-confirms-hack-over-60-of-stolen-passwords-already-cracked/`: of about 5.8 million leaked SHA-1 password hashes, over 60% could be brute-forced within a day or two.

is secure, for example if she is typing directly into the host computer's keyboard[9]. It is obviously hopelessly insecure if Alice is trying to authenticate to a remote host over an insecure network, as any eavesdropper can see her password and use it themselves later (the **replay attack**). The old **File Transfer Protocol** (FTP) used a cleartext password like this, and was a serious security problem.

Lamport proposed an elegant scheme which defeats an eavesdropper's replay attack, although it is only secure against passive attackers and cannot protect Alice if her communications are being redirected via an intruder. At initialization, a large integer $n$ is chosen and $\langle \text{USERNAME}_A, h^n(\text{PASSWORD}_A)\rangle$ written to the password database. $h^n(x)$ indicates that the function $h$ is iterated $n$ times on $x$. Alice must remember the integer $n$. Then the authentication protocol is

1. $A \rightarrow H$ : $\text{USERNAME}_A$
2. $H \rightarrow A$ : `Code?`
3. $A \rightarrow H$ : $\text{CODE} = h^{n-1}(\text{PASSWORD}_A)$
4. $H$ grants access if $\langle \text{USERNAME}_A, h(\text{CODE})\rangle$ is in its password database and if so **updates** the password entry to $\langle \text{USERNAME}_A, \text{CODE}\rangle$.
5. $A$ decrements $n$.

Once a correct code has been sent, it immediately becomes out of date and cannot be used by an eavesdropper. This protocol has the disadvantage that user must keep their $n$ in sync with what is expected by the host: a lost confirmation from the host could cause all future attempts to fail, for example. It also has the disadvantage, though some might say advantage, that the password must be changed every $n$ uses.

### 4.3.3 Authentication Tokens and Biometrics

Instead of, or as well as, a password, some systems require a physical device called an **authentication token**. The device could be a magnetic swipe card, a smart card (a tiny device with embedded processor), or just a car key. Such devices are more difficult to clone than eavesdropping on a password, and the more complex the device the more difficult the cloning. Car keys are easy to copy, but smart cards protected by a PIN and offering cryptographic challenge-responses are much more difficult. Online banking

[9]...and knows that the host has not had a keylogger installed, and knows that the keyboard and keyboard cable have not had a keylogger attached. And knows that nobody is standing behind her, etc.

often requires the use of a standalone, PIN-protected, calculating machine (of which more later).

Finally, if a password is *something you know* and a token *something you have*, a biometric is *something you are*: a fingerprint, retinal scan, voice ident, etc. Biometrics are often difficult to clone (particularly without connivance of the owner), but unlike passwords and tokens they are difficult or impossible to change.

Sections 10.8 to 10.10 of the course text (Kaufman et al., 2002) cover this material very well, and it will not be repeated here.

### 4.3.4 The LM Hash

Here we consider a short case study. The LM Hash, which is 128 bits long, was used to store passwords in versions of Microsoft Windows up to the early 1990s. It is a good lesson in bad hash function use because it manages to reduce the work of brute-force search from $2^{128}$ to about $2^{44}$. Sadly, it continued to be supported for reasons of backwards compatibility and only the most recent versions of Windows disabled it by default. The algorithm is as follows:

**LM Hash**

(1) Null-pad the password to 14 characters. (Longer passwords were not allowed).
(2) Force any lowercase letters to uppercase, to make the password case insensitive.
(3) Split into two 7-byte halves, each of which makes a 56-bit DES key.
(4) Use each DES key to encrypt the fixed 64-bit block consisting of ASCII characters `KGS!@#$%`.
(5) Concatenate the two DES encryptions to make a 128-bit hash.

Guessing a preimage – a password which hashes to a given value – should involve searching $2^{128}$ possibilities, but a restriction to 14 printable ASCII characters reduces this to about $95^{14} \approx 2^{92}$. Making it case insensitive further reduces the search space to 69 characters, $69^{14} \approx 2^{86}$ possibilities. The worst part of the algorithm is producing two separate hashes of seven-character halves: this way, one can brute force each half separately for a total work of about $2 \cdot 2^{43}$. (And most likely the second half of the password is not the full seven characters, which means that it will probably be found more quickly, and then these characters might be used to help with a dictionary attack on the first half).

There is no salt, so the same password leads to the same LM hash on every Windows system. Although a full table of hash inverses would still be too large to store on a modern-day computer, it is feasible to compute and store a rainbow table or similar structure, and these now can be found on the internet. Any LM hash can now be inverted within a few seconds.

## 4.4 Hashes and Key Generation

Recall from chapter 3 the need for Alice and Bob to share a secret cipher key. It might not be too easy to them to memorise a binary sequence, of length 56 for DES or 128 for AES, so they will want a way to generate the key from a more memorable secret. Such a device is called a **key derivation function** (KDF) and it can be used in many situations, including when Alice and Bob want to generate a disposable key from a secret they already share.

Key generation is an important part of a cryptosystem, which we did not discuss in chapter 3, and it seems to be an area which has received relatively little attention. We can only present some simple techniques here, and urge caution[10].

Obviously Alice and Bob cannot simply hash their memorable password to use as a key, because that exposes them to dictionary attacks: the enemy can try a KPA using hashes of extended dictionaries and have a good chance of much quicker success than plain exhaustion over the key. Since we are avoiding dictionary attacks, the same techniques that were used in the previous section apply. Ideally we want the KDF to be so difficult to attack that it would be preferable to exhaust over the key itself instead of the shared password.

Alice and Bob should certainly add a salt to the hash, at least 64 bits long, but if we take Kerckhoffs' Principle seriously then we should not rely on keeping the salt secret. Their best defence is to make the KDF very slow, bearing in mind that Alice and Bob should not need to use it very often. The "stretching" technique is commonly used, with the hash being iterated many times, perhaps thousands, in order to slow down a dictionary

[10] An interesting history of password hash and KDF development can be found at `http://www.openwall.com/presentations/PHDays2012-Password-Security/`.

attacker. One example of a KDF like this is

$$
\begin{aligned}
u_1 &= h\Big((p \oplus c_1) \,\|\, h\big((p \oplus c2) \,\|\, s\big)\Big), \\
u_i &= h\Big((p \oplus c_1) \,\|\, h\big((p \oplus c2) \,\|\, u_{i-1}\big)\Big) \text{ for } i = 2, \ldots, n, \\
k &= \bigoplus_{i=1}^{n} u_n
\end{aligned}
$$

where $p$ is the shared password, $s$ a fixed salt, $c_1$ and $c_2$ constants. The idea is simply to XOR together hashes iterated $n$ times, where $n$ might be 1000 or more, and this process produces a key which is the same length as the hash function (it can be chopped down if a smaller key is needed). The nested hash is a **MAC** function which will be explained in chapter 6.

The standard key derivation function PBKDF2[11] slightly generalizes the above scheme to allow generation of keys which are longer than the length of the hash function. Its underlying hash function is SHA-1. It is used in the WPA and WPA2 wireless authentication protocols.

Although cryptographic hashes are used in KDFs, the application does not require collision resistance or 2nd-preimage resistance. Even preimage resistance is only important if Alice and Bob need to keep their original key, and their salt, secret from the attacker. The most important thing is that the iterated hash output should be well-dispersed over its codomain, but it is not difficult to see why this is a prerequisite for good collision resistance anyway.

## 4.5 Hashes and Message Integrity

As we stressed in chapter 3, encryption does not guarantee the **integrity** of the encrypted message. This is relevant in the context of an **active attacker** who might, maliciously, alter the contents of communications between Alice and Bob. Bob may need a mechanism to ensure that the message is not altered in transit, and to reject it (and presumably ask for a repeat transmission) if it has been.

A cryptographic hash function is useful because it can accept an arbitrary-length input. If Alice's message is $m$ she can use the hash $h(m)$ as a guarantor of integrity, because

[11] `http://www.ietf.org/rfc/rfc2898.txt`

a modified message $m'$ will no longer (with very high probability) have the same hash. However, a hash alone cannot provide integrity: the simple protocol

1. $A \to B : m||h(m)$
2. $B$ separates the message $m'||y$ and accepts $m'$ only if $h(m') = y$.

does not provide integrity. An attacker capable of replacing $m$ by $m'$ can likewise replace $h(m)$ by $h(m')$, fooling Bob into accepting $m'$.

Nonetheless, this is essentially what is going on when one downloads software from a website which provides a **checksum** (usually an MD5 hash, see subsection 4.6.3) of the package. The checksum is partly to identify non-malicious transmission errors, which it does perfectly well, but also to prevent an attacker from replacing the software with their own malicious code. However, it seems likely that an attacker could replace the hash on the website at the same time! And few people actually bother to check that their downloaded package has the correct checksum.

Proper message integrity requires a key, similar to an encryption key. One solution is

1. $A \to B : E_k(m||h(m))$
2. $B$ decrypts and separates the message $m'||y$, accepting $m'$ only if $h(m') = y$.

where $E_k(-)$ is some secure encryption and $k$ a shared secret key. This prevents an attacker from replacing $h(m)$ with the new hash. This paradigm is used often, but it is unsatisfactory for two reasons. First, the message integrity relies on its confidentiality, and we would prefer to separate the security concerns. Second, it is vulnerable to a **truncation attack**: if Alice can be persuaded to send $m_2 = m \parallel h(m) \parallel m_1$ to Bob, their transmission is $E_k(m \parallel h(m) \parallel m_1 \parallel h(m_2))$ which, under some block cipher modes, can be truncated by the attacker to $E_k(m \parallel h(m))$. Thus the attacker fools Bob into thinking that Alice intended to send $m$, but this was only *part of* the message whose integrity was supposed to be preserved.

Another solution is

1. $A \to B : m \parallel E_k(h(m))$
2. $B$ separates $m'||c$, accepting $m'$ only if $h(m') = D_k(c)$.

This is an example of a Message Authentication Code (MAC), which we shall examine in Chapter 6. Note that, in this protocol, the message can be sent in the clear.

### 4.5.1 Do Collisions Matter?

When used in this way, the 2nd-preimage resistance property of the cryptographic hash function $h$ is paramount: it is what prevents the attacker from finding a different message $m'$ which the same hash. But cryptographic hash functions require full collision resistance, which would mean that an attacker cannot find *any* pair of messages with the same hash. An attack on collision resistance would probably find a pair of random-looking byte sequences which happen to hash to the same value: is this really a problem? We have shown in section 4.2 that (strong) collision resistance is a difficult condition requiring, all other things being equal, hash outputs to be twice as long for an equivalent level of security against exhaustion attacks. Perhaps 2nd-preimage resistance is sufficient, and we can dispense with full collision resistance?

This is a dangerous way of thinking, particularly for commonly-used hash functions. It is a consequence of their iterative construction that common hash functions have the property

$$h(x) = h(x') \text{ implies } h(x \,\|\, y) = h(x' \,\|\, y)$$

for all $y$. This is true of MD5 and SHA-1, the most widely-used hash functions today, if they are not padded correctly. There have been a number of attacks which find, by brute force or more clever methods, collisions in MD5 which amount to (abstracting all the details)

$$h(\texttt{v=0;}) = h(\texttt{v=1;})$$

from which one can produce code (or documents, or certificates...) which satisfy

$$h(\texttt{v=0;if v==0 then ...something innocent... else ...something guilty})$$
$$= h(\texttt{v=1;if v==0 then ...something innocent... else ...something guilty})$$

See (Daum & Lucks, 2005) for a readable description of this attack, with links to other work. The moral of the story is: any hash collision is potentially an attack on integrity.

## 4.6 Construction of Cryptographic Hash Functions

Finally, we outline how cryptographic hash functions are constructed in practice. This field is not as mature as cryptography, and most experts fear that the current options

have significant weaknesses, but there is hope that newly-developed hash functions will be as secure, in their own way, as AES seems to be for symmetric key cryptography.

### 4.6.1 The Merkle-Damgård Construction

We have already seen, in subsection 4.1.2, that the compression property of hash functions can be achieved by iterating a compression function over fixed-length blocks of hash input:

$$h\langle\rangle = IV$$
$$h\langle x_1, x_2, \ldots, x_m\rangle = g(h\langle x_1, \ldots, x_{m-1}\rangle, x_m)$$

for some initialization vector $IV$ (which must be fixed).

Hashes computed in this way are sometimes strengthened by applying one more function to the hash output. This is called **finalization**.

This iterative construction was studied independently by Ralph Merkle and Ivan Damgård, who proved that this is a secure construction in the sense that if $g$ is a compression function in the sense of section 4.1 **and** the input $x$ is padded up to a multiple of the blocksize in a secure way, then $h$ will be a cryptographic hash function. It is now known as the **Merkle-Damgård construction** and most hash functions use it. The security result can be made quite precise, in terms of measuring the "advantage" gained by an attacker (which, loosely speaking, is the probability that an attacker can break the one-way or collision-resistance properties): the maximum advantage against the iterated construction $h$ is no greater than the maximum advantage against the compression function $g$.

**Padding** the input turns out to be extremely important for security, because the iterative nature of the construction is open to abuse. For example, imagine a compression function with block length 8 bytes. If we simply pad out the input with trailing zeros, then

$$h(\texttt{message}) = h(\texttt{message0})$$

(where `0` represents a zero byte) is an easy-to-find collision. This is avoided by padding with `100`$\cdots$, which may require adding an additional block.

A weak flaw can exist if any padded message can ever be a suffix of another, i.e. $\textsc{pad}(x) = y \,\|\, \textsc{pad}(x')$ for some $x$, $x'$, and $y$. In such a case it is possible for collisions to be found amongst padded messages, even when $h$ is collision resistant. A proof of this highly technical result can be found in (Nandi, 2009). Motivated by this, it is usual to encode

the size of the message in the padding. Merkle's original suggestion, now called **MD-strengthening**, uses the padding

$$\text{PAD}(x) = x100\cdots0 \,\|\, l$$

where $l$ is the binary representation of the length of $x$ (measured in bytes or bits; this limits the possible input to a size which can be encoded within one block). Thus as many as two additional blocks might be added to the original data.

To prove security of the overall hash function, the padding must preclude these attacks. So called **MD-compliant** padding is a sufficient condition:

(i) $x$ is a prefix of $\text{PAD}(x)$.
(ii) If $|x| = |y|$ then $|\text{PAD}(x)| = |\text{PAD}(y)|$
(iii) If $|x| \neq |y|$ then the last blocks of $\text{PAD}(x)$ and $\text{PAD}(y)$ are not equal.

It is easy to see that the "MD-strengthening" padding satisfies these conditions.

We should say that there are arguments against the MD construction. Joux (Joux, 2004) points out that collisions of two messages can be extended, exploiting the iterative structure, to make collisions of exponentially many (unpadded) messages. In (Ferguson et al., 2010) the authors are very critical about the use of iterated compression functions: they claim that the intermediate values of the computation give away information which makes attacks more likely. The advocate using $h(h(x) \,\|\, x)$, instead of $h(x)$, for any iterative hash function $h$. This, however, takes twice as long to compute.

### 4.6.2 The MD4 Hash

The cryptographic hash function called MD4 was published by Ron Rivest in 1990[12]. It was designed to be fast on 32-bit processors, and we will use "word" to mean a 32-bit quantity throughout this section. The input is processed in blocks of 512 bits (16 words). The output is 128 bits (4 words), which is perhaps too short: recall that the birthday paradox means that only about $2^{64}$ hashes would be needed to find a collision by chance. MD4 follows the Merkle-Damgård paradigm, with a fixed IV and the following compression function written as a function of 4+16 input words and 4 output words[13]:

$$g(h_1, h_2, h_3, h_4, x_1, \ldots, x_{16}) = (h_1 + h'_1, h_2 + h'_2, h_3 + h'_3, h_4 + h'_4)$$

[12] http://www.ietf.org/rfc/rfc1186.txt

[13] This description is to give a flavour of the algorithm, not the full details; see (Menezes et al., 1996) for a complete description. We have not followed the description of (Kaufman et al., 2002) because it is incomprehensible.

where $(h'_1, h'_2, h'_3, h'_4)$ is the result of iterating 48 rounds of the function:

$$r(a, b, c, d) = (d, F_j, b, c)$$

to starting values $(h_1, h_2, h_3, h_4)$. This looks rather like a 48-round Feistel network, except that there are four blocks of bits which are shuffled around. (Remember though that cryptographic hash functions are not supposed to be invertible!) $F_j$ is a "mangler" function output, which is slightly different in each round, defined by

$$F_j = (a + f_j(b, c, d) + x[i_j] + c_j) \lll s_j$$

where

- $j$ is the round number (1–48);
- $f_j$ is a nonlinear binary function (operating bitwise on three 32-bit inputs), it changes every 16 rounds;
- $i_j$ determines which of the sixteen 32-bit data words of $x$ is used in this round (each is used three times);
- $c_j$ is a constant which changes every 16 rounds;
- $s_j$ is the amount of rotational left bit shift, which changes every round;
- The operator $\lll n$ rotates the 32 bits in a word $n$ to the left;
- $+$ represents addition modulo $2^{32}$.

MD4 throws just about everything at the bits in an effort to smash any structure and diffuse any changes. It represents a triumph of overcomplexity because it is *completely insecure* in spite of all this. Only five years after its proposal in 1990, the first full collision attack on MD4 was found (Dobbertin, 1998). It reduced the work required from $2^{64}$ MD4 computations to the equivalent of $2^{20}$, and thus was able to find two distinct messages with the same hash within a few seconds on a PC. Subsequent work found and refined new attacks to the point where only three MD4 hash operations were required in order to calculate a colliding pair of inputs: a comprehensive refutation of its collision-resistance. Furthermore, Guo et al. claim preimage attacks which reduce the work required for finding a preimage (respectively 2nd-preimage) from $2^{128}$ to about $2^{78}$ (respectively $2^{69}$) MD4 operations (Guo et al., 2010). While still a large amount of computation, this suggests that the one-way property of MD4 cannot be relied upon.

### 4.6.3 Successors to MD4

MD4 was popular for a while; it was used, for example, in the NTLM password digest (NTLM is the successor to LM from subsection 4.3.4). But even before the collision and

preimage attacks there were doubts about its security: it is common, in both cryptography and hashing, to find attacks on **reduced-round** versions before a full attack, and a 32-round version of MD4 was attacked as early as 1991. We now briefly sketch the widely-used successors to MD4.

**MD5** was published by Ron Rivest in 1992 and became very widely used, particularly to check the integrity of file transfers in the days when modems where the most common method of communication. MD5 "checksums" are still used in some file repositories today.

MD5 is not all that different to MD4. It adds 16 additional rounds to the iteration in the compression function, changes the round function to

$$r(a, b, c, d) = (d, b + F_j, b, c),$$

and then replaces the nonlinear functions $f_j$, the indexes $i_j$, the constants $c_j$, and the rotational shifts $s_j$ (mainly by giving them a less regular pattern). Everything else is the same. The full details can be found in (Menezes et al., 1996, §9.4.2(ii)) and many other places online.

The security of MD5 lasted longer than MD4, with collisions created in 2004 and a series of more serious attacks then following. The current best attack takes about $2^{21}$ computations to create a collision. At the present time, there are no preimage or 2nd-preimage attacks significantly better than exhaustive search. Nonetheless, it is considered an insecure hash and new security systems will avoid it.

MD5 is used by some certification authorities to validate SSL certificates, and it has been shown that collisions can be exploited to create fake validation of invalid certificates. In 2012 an MD5 collision was used, by the widespread malware *Flame*, to forge a Windows code-signing certificate.

**SHA-1** was published in 1995. Like DES it was published by NIST as a US Federal Information Processing Standard. It was designed by the NSA, which immediately made some people suspicious of it. Almost every modern application which does not use MD5 uses SHA-1 (and some use both).

SHA-1 has a hash length of 160 bits. It is similar to MD5 but feeds five, rather than four, 32-bit words from compression function to compression function. Each compression

function iterates 80 rounds,

$$r(a, b, c, d, e) = (F_j, a, b \lll 30, c, d)$$

and the mangler functions $F_j$ are rather similar to those in MD4. The main difference is that each message block is expanded to a full 80 32-bit words, by forming different XOR and bit-rotated combinations of the 16 32-bit words which make up the 512-bit block. These 80 words are then used in one round each. The full details can be found in (Menezes et al., 1996, §9.4.2(iii)).

There is some uncertainty about the best attacks on SHA-1, perhaps because they are so complicated that it takes a long time to verify them. There are, as yet, no preimage or 2nd-preimage attacks better than exhaustive search, but claims have been made that collisions can be found in $2^{50}$–$2^{60}$ hash computations, compared with the birthday paradox baseline of $2^{80}$. Some attacks on reduced-round SHA-1 make cryptographers nervous about the security of the full version.

There is also a **SHA-2**, which can output hashes of 224–512 bits. It is not widely used, perhaps because SHA-1 weaknesses were not identified until 2005, and are still not very severe. A 5-year contest to select **SHA-3**, inspired by the contest structure which produced the AES encryption algorithm, finished in October 2012; it was not held because of weaknesses in SHA-2, but in order to have a differently-designed alternative available (in case of future attacks). The algorithm known as **Keccak** was the winner. In its default 64-bit mode (in which all operations are on 64-bit words, for efficiency on modern processors) it supports outputs of 224, 256, 384, and 512 bits; its word size can also be scaled up or down. Hopefully these super-secure hash functions will begin to be adopted.

## Bibliography

Daum, M. & Lucks, S. (2005). Hash Collisions (The Poisoned Message Attack). Webpage based on Eurocrypt 2005 rump session talk: `http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/`.

Dobbertin, H. (1998). Cryptanalysis of MD4. *Journal of Cryptology*, 11, 253–271. Available at `http://repo.meh.or.id/Cryptography/EN-Cryptanalyse%20Md4.pdf`.

GUO, J., LING, S., RECHBERGER, C., & WANG, H. (2010). Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science* (pp. 56–75). Springer. Extended version available at http://eprint.iacr.org/2010/016.pdf.

HELLMAN, M. E. (1980). A cryptanalytic time–memory trade-off. *IEEE Transactions on Information Theory*, 26(4), 401–406. Available at http://www-ee.stanford.edu/~hellman/publications/36.pdf.

JOUX, A. (2004). Multicollisions in iterated hash functions. Application to cascaded constructions. In *Advances in Cryptology – CRYPTO 2004, 24th Annual International Cryptology Conference*, volume 3152 of *Lecture Notes in Computer Science* (pp. 306–316). Springer. Available at http://www.iacr.org/cryptodb/archive/2004/CRYPTO/1472/1472.pdf.

KNUTH, D. E. (1998). *The Art of Computer Programming Volume 2 – Seminumerical Algorithms*. Addison-Wesley, 3rd edition.

NANDI, M. (2009). Characterizing padding rules of MD hash functions preserving collision security. In *Information Security and Privacy*, volume 5594 of *Lecture Notes in Computer Science* (pp. 171–184). Springer. Available at http://eprint.iacr.org/2009/325.pdf.

# Chapter 5

# Asymmetric Key Ciphers

---

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §2.5, 6.1-6.3, chapter 7 |
| **Alternatives & further depth:** | Gollmann §14.2, 14.5.3 (barely covered) |
| | Ferguson et al. §2.3, chapters 10 & 12 |
| | Menezes et al. §2.4.1-2.4.3, 4.1, 4.2.3, 8.1-8.2, 11.3.6 |

---

A major drawback of symmetric key ciphers is the requirement for a secret key. Although it might be feasible for small numbers of people to meet, and for each pair to agree on their own key, for larger groups it becomes both unwieldy to keep track of a key for every pair of people and impossible to exchange the keys in the first place. Asymmetric ciphers allow users to communicate confidentially after transmitting a key which is not confidential[1]. Intuition suggests that this ought to be impossible, because an eavesdropper knows everything the participants transmit, but it becomes possible with mathematical tricks called **trapdoor functions**. A trapdoor function is a bit like a hash in that it is easy to perform and difficult to invert, but different in that it becomes easy to invert given some extra information. The RSA system uses a classic example and we will dedicate this chapter to it.

With an asymmetric cipher, the encryption key and decryption key are different. The encryption key, called the **public key**, is published, and it should be computationally

[1]The authenticity of the public key, however, does need to be established. We return to this in chapter 7.

infeasible to deduce the decryption key, called the **private key** or **secret key**, from the public key. This makes key distribution much easier, because everyone can publish their own public key. (In the following chapter we will see how to create asymmetric signatures, which tend to go hand-in-hand with asymmetric ciphers, to prevent an attacker from forging another person's key; later in the course we can combine all the components to allow users who have never met to verify each others' identity and communicate securely.)

## 5.1 Number Theory Primer

We summarise, very briefly, the necessary number theory to understand why the RSA cryptosystem works.

- The notation $x \mid y$, "$x$ **divides** $y$", means that the integer $y$ is a multiple of the integer $x$.
- A positive integer $p$ is **prime** if its only divisors are 1 and itself. (1 is not a prime number).
- The **greatest common divisor** of two integers, $m$ and $n$, is the greatest integer which divides both. It is written $\gcd(m, n)$.
- $\gcd(m, n)$ can be computed extremely efficiently using Euclid's Algorithm. Furthermore, if $\gcd(m, n) = g$ then Euclid's Algorithm finds integers $x, y$ satisfying the equation $g = mx + ny$. Euclid's Algorithm takes $O(\log \min(m, n))$ steps and each step involves a fixed number of arithmetical operations on numbers no greater than $m$ or $n$.
- Two integers, $m$ and $n$, are **coprime** if $\gcd(m, n) = 1$.
- When $\gcd(m, n) = 1$,
$$\text{if } m \mid x \text{ and } n \mid x \text{ then } mn \mid x. \tag{5.1}$$
- We write $x \equiv y \pmod{n}$ if $n \mid x - y$, which is equivalent to saying that $x$ and $y$ have the same remainder when divided by $n$. The positive integer $n$ is called the **modulus**.
- Arithmetic can be performed on the remainders modulo $n$, because $x_1 \equiv y_1 \pmod{n}$ and $x_2 \equiv y_2 \pmod{n}$ imply that $x_1 + x_2 \equiv y_1 + y_2 \pmod{n}$, $x_1 \cdot x_2 \equiv y_1 \cdot y_2 \pmod{n}$, and $x_1^e \equiv y_1^e \pmod{n}$ for any positive integer $e$.
- If $\gcd(a, n) = 1$, there exists a **multiplicative inverse** for $a$, which is an integer $b$ such that $ab \equiv 1 \pmod{n}$. When $n$ is implicit, the inverse of $a$ is written $a^{-1}$. It can be found using Euclid's Algorithm: we find integers such that $ax + ny = 1$, and then $a^{-1} \equiv x \,(\text{mod } n)$.

Most textbooks present quite a lot of number theory in order to get to RSA. We only need one result, a special case of **Euler's theorem** often known as **Fermat's Little Theorem**:

**Claim 5.1** For prime $p$,

$$a^{p-1} \equiv \begin{cases} 0 \pmod{p}, & \text{if } p \mid a, \\ 1 \pmod{p}, & \text{otherwise.} \end{cases} \tag{5.2}$$

**Proof** If $p \mid a$ then $p \mid a^{p-1}$, so the first case is immediate. Suppose that $p$ does not divide $a$. Now consider the following sets,

$$A = \{1, 2, \ldots, p-1\},$$
$$B = \{a \cdot 1 \pmod{p}, a \cdot 2 \pmod{p}, \ldots, a \cdot (p-1) \pmod{p}\}.$$

No members of $B$ can be zero since $a \cdot x \equiv 0 \pmod{p}$ requires $p \mid a$ or $p \mid x$, neither of which happens. All members of $B$ must be different since $a \cdot x \equiv a \cdot y \pmod{p}$ requires $p \mid a \cdot (x-y)$ which means $p \mid a$ or $p \mid x-y$, neither of which happens. By the pigeonhole principle, $A$ and $B$ must be the same set. Now multiply the elements in each set together: we must have

$$\prod_{i=1}^{p-1} i \equiv \prod_{i=1}^{p-1} (a \cdot i) \equiv a^{p-1} \prod_{i=1}^{p-1} i \pmod{p}. \tag{5.3}$$

Finally, $\prod_{i=1}^{p-1} i$ is coprime to $p$ so it has a multiplicative inverse $b$. Multiplying (5.3) by $b$ gives the desired result. ∎

## 5.2 The RSA Cryptosystem

The RSA system is named after its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman, and was first published in 1978. In fact, it had already been discovered by Clifford Cocks at GCHQ in 1973, but was considered more of a mathematical curiosity than a practical cipher. Mathematical work at GCHQ was classified and not published, and it was not until 1997 that Cocks was allowed to admit his earlier discovery in public. Until 2000 there were some restrictions on the RSA algorithm because it was patent encumbered. It is now used very widely in cryptographic and authentication protocols.

The keys are generated entirely by the recipient Bob. He first chooses a key length $b$: larger $b$ increases the security of the system and also increases the length of message which can be sent, but at greater computational cost. A typical value is $b = 1024$.

**Key Generation**

(1) Choose two prime numbers each $b/2$ bits in size, call them $p$ and $q$. Compute their $b$-bit product $n = pq$ and the so-called **totient** $\phi = (p-1)(q-1)$.
(2) Choose an integer $e$ which is coprime to $\phi$.
(3) Publish the public key, which is the pair $pk_B = \langle n, e \rangle$.
(4) Find an integer $d$ such that $ed \equiv 1 \pmod{\phi}$.
(5) The private key is the pair $sk_B = \langle n, d \rangle$[2].

Alice can use the public key $pk_B$ to transmit a message, confidentially, to Bob. In the terminology of chapter 3, the keyspace refers only to the unpublished part of the secret key, the integer $d \in \mathcal{K} = \{2, \ldots, n-1\}$. The plaintexts and ciphertexts are both numbers and $\mathcal{M} = \mathcal{C} = \{0, 1, \ldots, n-1\}$.

The encryption and decryption functions use the public and private keys, respectively:

**Encryption**

(1) ("pad" $m$, which we will later see is necessary for security).
(2) $E_{pk_B}(m) = m^e \pmod{n}$.

**Decryption**

(1) $D_{sk_B}(c) = c^d \pmod{n}$.
(2) (discard padding).

The RSA system is making use of the following trapdoor: given $n$ and $e$, it is easy to compute $m^e \pmod{n}$ from $m$ but difficult to do the reverse; however, given the factors of $n$ it is easy to follow the mathematics below and find $m$ from $m^e \pmod{n}$.

So much for the abstract formulation. We need to check that decryption actually works, that the above algorithms can be performed efficiently, and that an attacker cannot decrypt in reasonable time without the secret key.

[2] If Bob stores $p$ and $q$ as well, he can make decryption somewhat more efficient. See (Kaufman et al., 2002, §6.3.4.4).

### 5.2.1 Correctness

We need to check that $D_{sk_B}(E_{pk_B}(m)) = m$. Because the message is smaller than $n$, it suffices to prove that this holds (mod $n$).

The encryption and decryption algorithms give that $D_{sk_B}(E_{pk_B}(m)) \equiv m^{ed} \pmod{n}$. And the key generation algorithm ensured that $ed \equiv 1 \pmod{\phi}$, which is equivalent to $ed = 1 + k(p-1)(q-1)$ for some integer $k$. Now calculate

$$m^{ed} = m \cdot m^{k(p-1)(q-1)} \equiv m \cdot (m^{p-1})^{k(q-1)} \overset{(5.2)}{\equiv} \left\{ \begin{array}{ll} m \cdot 0^{k(q-1)}, & \text{if } p \mid m \\ m \cdot 1^{k(q-1)}, & \text{otherwise} \end{array} \right\} \equiv m \pmod{p},$$

$$m^{ed} = m \cdot m^{k(p-1)(q-1)} \equiv m \cdot (m^{q-1})^{k(p-1)} \overset{(5.2)}{\equiv} \left\{ \begin{array}{ll} m \cdot 0^{k(p-1)}, & \text{if } q \mid m \\ m \cdot 1^{k(p-1)}, & \text{otherwise} \end{array} \right\} \equiv m \pmod{q}.$$

Now we have $p \mid m^{ed} - m$ and $q \mid m^{ed} - m$, which with (5.1) gives $pq \mid m^{ed} - m$ i.e.

$$m^{ed} \equiv m \pmod{pq}.$$

(Some literature incorrectly asserts that it is necessary for $m$ to be coprime to $n$ for the above result to hold. It is necessary for a particular proof to work, but our proof works for any $m$.)

### 5.2.2 Efficiency

All parts of the RSA cryptosystem depend on modular arithmetic with a very large base, $n$ or $\phi$. The numbers involved will easily overflow 32- or 64-bit integers so they must be stored in some sort of large or arbitrary-precision number format. Because these formats are not native to CPUs, operations on them will be slow. We must use efficient algorithms to implement RSA.

The key generation phase begins with searching for very large prime numbers in a certain range (with a certain number of bits). Stage (1) will be done using a pseudo-random primality test, for example the Miller-Rabin test. Depending on the exact algorithm used, the time complexity will be $O(b^3)$ or $O(b^4)$. In stage (2), we choose $e$ and then check that $\gcd(e, \phi) = 1$ using Euclid's Algorithm (if not, repeat with a different value of $e$, or if $e$ is fixed we regenerate $p$ and $q$). This takes $O(b)$ steps, each involving a fixed number of addition and multiplication operations on $b$-bit integers, for an overall time of $O(b^3)$. The same run of Euclid's Algorithm gives us $d$ as well, since it is the multiplicative inverse of $e \pmod{\phi}$ and that completes all the computation of the key generation phase.

We can make the encryption more efficient by choosing a fixed small value of $e$: the only constraint is that it must be coprime to $p-1$ and $q-1$. Obviously $e$ should not be 1, otherwise encryption is the identity function, and $e = 2$ is impossible because it will not be coprime to the even numbers $p-1$ or $q-1$. It is quite popular to choose $e = 3$, and search for primes $p$ and $q$ such that $p-1$ and $q-1$ are not multiples of 3. This allows for very fast encryption, using only two modular multiplications[3]. Or sometimes $e = 65537$ is chosen. However we cannot choose a convenient exponent for decryption because $d$ **must** be kept secret so cannot be fixed. As the solution to the equation $ed \equiv 1 \pmod{\phi}$, $d$ will typically be $b$ bits long.

When it comes to exponentation in encryption or decryption, we make use of the **method of repeated squaring**,

$$x^y = \begin{cases} 1, & \text{if } y = 0 \\ (x^2)^k, & \text{if } y = 2k \\ x \cdot (x^2)^k, & \text{if } y = 2k+1 \end{cases}$$

which allows us to compute exponentation to power $y$ using $O(\log y)$ modular multiplications. In our case, if $e$ is fixed then $O(1)$ multiplications are used for encryption and $O(b)$ for decryption. Each modular multiplication is of $b$-bit numbers, which normally takes $O(b^2)$ CPU operations (including the division-and-remainder operation to do the modular reduction). Overall then, encryption with a fixed $e$ takes $O(b^2)$ operations, and decryption takes $O(b^3)$.

There are many algorithmic improvements, surveyed in detail in (Menezes et al., 1996, Chapter 14). A clever trick called Montgomery multiplication allows modular multiplication without a division step for the modular reduction (it is described concisely in (Guan, 2003)). The Chinese Remainder Theorem allows the multiplications to be done mod $p$ and $q$ separately, saving a factor of 4 on the time complexity (Kaufman et al., 2002, §6.3.4.4). For large $b$ (in practice for $b$ greater than about 1024), even the multiplication operation can be improved, for example by Karatsuba multiplication, which is $O(b^{\log_2 3})$ rather than $O(b^2)$ (Knuth, 1998, §4.3.3). There are a number of other possibilities, most of which are only useful for very large values of $b$.

It is commonly stated that RSA operations are very expensive, compared with symmetric key cryptography which is often designed precisely for CPU primitives. For example,

[3] Note that there are severe attacks on RSA if $e$ is small, and sometimes small $e$ is banned by industry standards, but these attacks are foiled by sensible padding and small $e$ is now generally considered safe. See section 5.4.

"DES and other block ciphers are much faster than the RSA algorithm. DES is generally at least 100 times as fast in software and between 1,000 and 10,000 times as fast in hardware, depending on the implementation."[4] In order to pin this down, the lecturer implemented both DES and RSA in C, and ran them on a 3.46GHz Xeon processor. The DES implementation was tightly-written, making use of the processor's 64-bit registers. The RSA implementation used a well-optimized library for manipulation of arbitrary-size integers, with varying $b$ and fixed $e = 3$. Plaintexts were random. The operations were benchmarked as follows:

| | $b = 512$ | $b = 768$ | $b = 1024$ | $b = 1536$ | $b = 2048$ | $b = 3072$ | $b = 4096$ |
|---|---|---|---|---|---|---|---|
| Key generation | 3 ms | 11 ms | 30 ms | 106 ms | 320 ms | 1123 ms | 3960 ms |
| Encryption | 1 $\mu$s | 3 $\mu$s | 7 $\mu$s | 13 $\mu$s | 24 $\mu$s | 46 $\mu$s | 85 $\mu$s |
| Decryption | 114 $\mu$s | 446 $\mu$s | 1.2 ms | 3.5 ms | 8.6 ms | 25 ms | 63 ms |
| DES equivalent* | 0.8 $\mu$s | 1.2 $\mu$s | 1.6 $\mu$s | 2.4 $\mu$s | 3.2 $\mu$s | 4.8 $\mu$s | 6.4 $\mu$s |

*En/decryption of 64-bit block using DES: 100 ns, scaled to RSA blocksize

The table reflects the approximate RSA time complexities we previously described: key generation is between $O(b^3)$ and $O(b^4)$, encryption with fixed $e$ is slightly better than $O(b^2)$, and decryption is approximately $O(b^3)$. It also shows that RSA encryption with a fixed small value of $e$ is only a few times slower than DES encryption, but decryption is 100-10000 times slower. The time needed for RSA key generation is quite substantial for large values of $b$ – even a few seconds, which could severely impede a protocol supposed to run in real time – but very large keys would only be used for long-term purposes and therefore infrequently generated.

It is one thing to run RSA algorithms on a 3.46GHz Xeon processor, and quite another to run it on devices with limited computational power. The following benchmarks are from a mobile phone (with a 1GHz processor)[5]. The Java library which the phone uses for RSA operations is fixed for $e = 65537$, so we will expect it to be a bit slower than with $e = 3$, and one would expect slower performance with code written in Java rather than a lower-level language like C.

| | $b = 512$ | $b = 768$ | $b = 1024$ | $b = 1536$ | $b = 2048$ | $b = 3072$ | $b = 4096$ |
|---|---|---|---|---|---|---|---|
| Key generation | 226 ms | 864 ms | 11 s | 35.8 s | 55.7 s | 257 s | 582 s |
| Encryption | 1 ms | 3 ms | 6 ms | 14 ms | 25 ms | 56 ms | 95 ms |
| Decryption | 13 ms | 37 ms | 86 ms | 267 ms | 633 ms | 2.1 s | 4.9 s |

[4] `http://www.rsa.kz/node/rsalabs/node54d2.html?id=2215`

[5] Many thanks to Bangdao Chen for mobile phone and smart card benchmarks.

Clearly, mobile phones are not able to generate large RSA keys, and even small keys would cause an unacceptable delay if they had to be generated anew on each communication (some protocols may require this). Even with a fixed key, the RSA encryption operation would cause long latencies (and unacceptable battery drain!) if it used to encrypt every wireless or GSM packet, for example. Smart cards are even more limited in computational power: simply to encrypt ($e = 65537$) a single 1024-bit RSA communication takes nearly 4 seconds on a processor typical of chip-and-pin cards, a Teridian 73S1217 (up to 24Mhz, 2KB memory). This presents cryptographers with interesting challenges, trying to squeeze the maximum security out of the hardware capabilities.

The use of RSA for everyday encryption, then, is rather expensive, and this is something true of all current asymmetric cryptosystems. What happens in practice is that RSA is used at the start of communication, just once or twice, to send or agree on a unique **session key**, a symmetric cipher key which is used for one series of communications and then discarded. (At the same time the communicating parties probably authenticate each other, using the same asymmetric cipher.) The session key can be quite long, if it fits within the RSA modulus, and the subsequent communications use much faster symmetric cryptography operations. In chapter 7 we will examine the protocols which do this.

### 5.2.3 Security

The encryption is only secure if an enemy is unable to deduce the plaintext $m$, which is to solve the so-called **RSA Problem**: from the ciphertext $m^e \pmod{n}$, and given $n$ and $e$, deduce $m$. As with symmetric ciphers, we can only appeal to the "Fundamental Tenet of Cryptography": the problem is well-studied by the academic community, and no computationally feasible solution has been found for large key sizes. Therefore an attacker should not be able to do it either.

We should be clear about these mathematical assumptions. The so-called **RSA Assumption** is that the RSA Problem is computationally infeasible for large $n$[6]. We will generally accept the RSA Assumption, with some caution.

There are a number of ways the RSA Assumption could be broken. Easiest would be if the attacker could deduce the factors $p$ and $q$. From then, they could simply follow the key generation algorithm to compute $d$ and decrypt messages as easily as Bob. But it is usually assumed that the so-called **Integer Factorization Problem** is infeasibly difficult,

[6] More precisely, given $n$ a product of two primes $pq$, $e$ coprime to $\phi = (p-1)(q-1)$, a randomly chosen $M$ less than $n$, for any function $A(n, e, x)$ computable in polynomial time, the probability that $A$ gives the right answer $P[A(n, e, M^e \pmod{n}) = M] \leq \nu(\log n)$, where $\nu(-)$ is some negligible function.

for large enough $n$. Integer Factorization is the most studied problem in computational number theory.

In fact the attacker does not need to know $p$ and $q$; it is enough to know $\phi$, from which they proceed to compute $d$ via $ed \equiv 1 \pmod{\phi}$. However, finding $\phi$ from $n$ is equally as difficult as factorizing $n$. You will prove this in one of the exercises. We can go one step further: the attacker does not even need to know $\phi$, since $d$ is enough. However, it is also proved that finding $d$ is computationally no easier than finding $\phi$, hence equivalent to factorizing $n$. You will prove this, at least in the case of small $e$, in the exercises.

If the Integer Factorization Problem were "solved" (in the sense that answers could be computed for much larger numbers than before), the security of RSA would be reduced or destroyed. But even if not, the RSA Problem could still be solved in its own right. All we know is that solving the RSA Problem is at least as easy as integer factorization[7]. The reference (Rivest & Kaliski Jr., 2005) is a very readable summary of the many variations on the RSA Assumption and their relevance to security.

If we accept the RSA Assumption, there are still dangers to avoid. We should not encrypt the plaintexts $m = 0$ or $m = 1$, since they encrypt to themselves under any key, we should not encrypt other small numbers because they might not cause any modular reductions, leading to an easy recovery of the plaintext, we should not encrypt the same message under many different keys, and all encrypted messages should be unpredictable: these can all be prevented by good padding and are discussed in section 5.4.

Just as with symmetric key cryptography, the most important thing we can do is to ensure a long enough key. But RSA keys will need to be a lot longer than their symmetric key counterparts.

So how long a key length is secure? Given proper padding, the most realistic attack at present is to try to factorize $n$. Despite the discussion above, attacks on the RSA Problem directly have been much less successful than those on integer factorization, so the size of secure key is determined by the attacker's ability to factorize. And if we want our secrets to remain secret for, say, 30 years then we need to consider an attacker's ability to factorize on computers, and with algorithms, available in 2040!

For large numbers, the state-of-art factoring algorithm is the general number field sieve (GNFS, see (Pomerance, 1996) for a history of its development), a rather complicated

[7] This is a common source of confusion. It is not yet proven that solving the RSA Problem is as difficult as solving integer factorization.

algorithm which has time complexity roughly $O\big(\exp(1.93b^{1/3}(\log b)^{2/3})\big)$. This function grows faster than any polynomial in $n$, but slower than any exponential: a polynomial-time factorization algorithm would be a major advance in computational number theory[8]. Parts of the GNFS algorithm can be parallelized efficiently and factorizations of very large numbers are attacked by clusters of computers.

Researchers constantly compete for the integer factorization record; based on historial records, Brent made a prediction in 1999: that a 768-bit RSA key would be factorized in 2009, a 1024-bit key in 2018, a 2048-bit key in 2041, and a 4096-bit key in 2070 (Brent, 1999). So far the prediction has been accurate: the the current record is factorization of a 768-bit RSA key and it was announced in 2010.

Home computers, if equipped with an optimized GNFS algorithm (open source implementations exist), can currently factorize 512-bit integers in a few weeks, so key sizes as low as this are only suitable for casual security. Current advice[9] is that 1024-bit keys are suitable for corporate use and 2048-bit keys for extremely valuable long-term use. As with symmetric cryptography, it pays to be conservative in case unpredictable algorithmic advances are made.

## 5.3 RSA Signatures

RSA encryption and decryption commute, i.e. $D_{sk_B}(E_{pk_B}(m)) = E_{pk_B}(D_{sk_B}(m)) = m$. This allows Bob to use the algorithm backwards, announcing the pair $\langle m, s\rangle = \langle m, D_{sk_B}(m)\rangle$ for some payload $m$. Anyone else can verify that $E_{pk_B}(s) = m$ and thus Bob has proved the authenticity of $m$, and also made it impossible to repudiate $m$, because nobody else could compute such an $s$. This forms the basis of RSA signatures, which we discuss in the next chapter.

## 5.4 Attacks on RSA

There are a few attacks on RSA which are based on its number-theoretic properties rather than attempts to factorize $n$.

[8] Shor's algorithm (Shor, 1997) is a cubic-time factorization algorithm for quantum computers. As yet it has not proved very useful: in 2012, a research group at UC Santa Barbara issued a press release saying that they had built a quantum computer which managed to show that $15 = 5 \times 3$, with approximately 50% certainty.

[9] `http://www.rsa.kz/node/rsalabs/node6501.html?id=2218`

We have already said that the plaintext $m$ should not be 0 or 1, because they encrypt to themselves under any key. In fact any value of $m < \sqrt[e]{n}$ is insecure, because $c = m^e$ does not involve a modular reduction at all: it means that the attacker can simply compute the integer root $\sqrt[e]{c}$ to recover the plaintext. This is actually a very likely situation when $e$ is a small fixed value, such as $e = 3$ that we suggested earlier, because the most typical use of RSA is to convey either a symmetric key or a cryptographic hash, which are likely to be only 64, 128, or 160 bits long: much smaller than the cube-root of a 1024-bit modulus.

In any public key scenario the Chosen Plaintext Attack is always available, because the attacker can use the public key to make their own ciphertexts at will. So Alice must be careful not to send messages which come from a small set of possibilities. For example, she should not use it to send a short password to Bob, because the attacker can try an offline dictionary attack, encrypting all the possibilities until he finds a ciphertext to match the one Alice sent.

There is a more subtle danger with RSA, called a **same-message attack**. If the attacker knows that Alice is sending the exact same message to at least $e$ different people, i.e. encrypted under $e$ different public keys, even if the keys are all different it is possible to recover the message $m$ without breaking the keys. This appears on one of the problem sheets.

Finally, RSA has a curious mathematical property. Because $(m_1 m_2)^e = m_1^e m_2^e$, we have

$$E_{pk_B}(m_1) \cdot E_{pk_B}(m_2) = E_{pk_B}(m_1 \cdot m_2). \tag{5.4}$$

This is called the **homomorphic property** of RSA and it looks a bit like the linearity property we saw in chapter 3. It leads to a potential attack under the Chosen Ciphertext Assumption (CCA, section 3.2) which implies an extremely strong adversary: if the attacker receives encrypted message $c = m^e \pmod{n}$, they can multiply by the encryption of a random number $r^e \pmod{n}$ and ask the decryption oracle to decrypt $r^e c$, which gives them back $rm$. Finally, they can multiply by $r^{-1} \pmod{n}$ to recover $m$. There are other attacks which exploit the homomorphic property too.

Most of these can be avoided by **padding** the plaintext before encryption. Unlike with hashing, padding does more than round up the plaintext to a full "block", it also acts to blind the contents. There are a number of ways to pad RSA plaintexts, and one is described by the PKCS#1 set of standards[10]. Here we will describe **PKCS#1 v1.5**. It is slightly outdated, but it is easy to see why it achieves its aims.

[10] `http://www.ietf.org/rfc/rfc3447.txt`

Suppose that a payload has been created. First, compute the maximum number of whole bytes guaranteed to fit within the modulus, $B = \lfloor \log_2(n-1)/8 \rfloor$. The payload will be padded to $B$ bytes. Suppose that the payload is a bytestream PAYLOAD, $p \leq B - 11$ bytes long. The plaintext is the integer $m$ represented by the bytestream

00 02 PADDING 00 PAYLOAD

where PADDING is a random sequence of $B - p - 3$ *nonzero* bytes. After this padding, the ciphertext is $m^e \pmod{n}$ as usual (PKCS#1 v1.5 is designed to be used with $e = 3$). Because the padding is terminated by the zero byte before the payload it can be removed unambiguously after decryption.

The padding slightly reduces the capacity of each encryption. But it defeats the attacks we mentioned above, because

- we never encrypt 0 or 1;
- moreover, the high-order bytes 00 02 guarantee that $m > \sqrt[3]{n}$;
- even if Alice's message is known to be from a small set, there are still very many possibilities for the plaintext, given that there are at least 8 bytes of random padding;
- similarly, it is vanishingly unlikely the the same plaintext will be sent to multiple recipients;
- the padding bytes should balk attempts to use the homomorphic property to tamper with messages;
- furthermore, in the unlikely event of the existence of a decryption oracle it should at least refuse to disclose the decryption of incorrectly padded messages, preventing the CCA-based multiplicative attack[11].

Note that padding, in the case of asymmetric cryptography, is markedly different to padding of hashes: the latter has to be nonrandom so that users can compare hashes, but the former can and must be randomized in order to prevent dictionary attacks on short plaintexts. In public key cryptography, randomized padding is an essential part of the cryptosystem, not an optional extra. (In symmetric encryption, padding was so unimportant that we barely mentioned it.)

[11] One should be rather careful about protocols where messages are refused for incorrect padding as it can give rise to subtle attacks. An attack on RSA-PKCS#1, if incorrect padding can be detected by an attacker, can be found in (BLEICHENBACHER, 1998).

There is another class of attacks on RSA: so-called **side channel attacks** observe the running of the RSA algorithms and deduce information about the secret key. For example, **timing attacks** use variations in the running time of modular exponentiation and multiplication as the operands vary, and **power attacks** measure variations in the power drain of CPUs as the calculations change. Although it might seem unlikely that much information can be gleaned from these measurements, both types of attack have been implemented successfully in real-world scenarios, for example (Brumley & Boneh, 2003). Power attacks are particularly applicable to smart cards.

## 5.5 Other Asymmetric Ciphers

RSA is far from the only asymmetric cipher, and there are also asymmetric schemes which provide digital signatures but not encryption itself. We have chosen to focus on RSA because it is widely used, fairly easy to describe, and has interesting mathematical analysis.

Many options are surveyed in (Menezes et al., 1996, Chapter 8). Some popular asymmetric ciphers include:

### Rabin public-key encryption

Even simpler than RSA: Bob generates two large primes $p$ and $q$; they form the secret key. He publishes $n = pq$, which is the public key. Encryption is $E_{pk_B}(m) = m^2 \pmod{n}$. Decryption is therefore a matter of finding square roots $\pmod{n}$; this is relatively easy when the factorization of $n$ is known, as for Bob, but difficult otherwise.

In this case, when $n$ is a product of two primes, it is known that finding square roots is equivalent to factorization in computational complexity. Thus the Rabin system is provably equivalent to the problem of integer factorization, and *provably secure* under the assumption that factorization is difficult. Despite this, it is not much used in practice.

It has similar mathematical dangers to RSA, and good padding is essential.

### Merkle-Hellman knapsack encryption

The idea behind this scheme is to use an NP-complete problem, in this case the infamous 0-1 knapsack problem, so that the attacker has to solve an infeasible problem to attack the encryption. The secret key is a trapdoor which makes the instance of the knapsack

problem tractable. However, the scheme is **insecure** and a decryption key can be deduced from the public key in polynomial time. (Some other knapsack-based encryption schemes do seem to be secure, but cryptography based on NP-complete problems has generally had little success.)

### Elliptic Curve Cryptography (ECC)

A more recent invention making use of algebraic operations on **elliptic curves**. Algorithms such as RSA are still applied, but instead of integers modulo $N$ we use points on a certain type of curve $y^2 = x^3 + ax + b$, and instead of multiplication we use a strange operation that maps two points on the curve to another. The advantage of this rather bizarre method is that the public keys are considerably shorter, and the problem equivalent to factorization has fewer known good algorithms for solving it. In practice, ECC keys can be considerably shorter than RSA keys for what is believed to be equivalent security: a 224-bit ECC-RSA key is thought approximately as secure as a 2048-bit RSA key.

## 5.6 Key Exchange

As we have said, a typical use of asymmetric ciphers is to transmit a symmetric **session key**. There are other methods for Alice and Bob to agree on a secret symmetric key using only public communication, which in fact predates RSA. The earliest **key exchange protocol** is called **Diffie-Hellman Key Exchange** (DHKE). The participants agree, publicly, on a large prime $p$ and a number $g$[12]. Then Alice picks a random integer $s_a$, and Bob a random integer $s_b$, which they keep secret. The protocol goes as follows:

1. $A \to B : a = g^{s_a} \pmod p$
2. $B \to A : b = g^{s_b} \pmod p$
3. $A$ computes $b^{s_a} \pmod p$
4. $B$ computes $a^{s_b} \pmod p$

After the communications, Alice and Bob have both computed $g^{s_a s_b} \pmod p$, which they can use to generate a session key.

Note that this is not a cipher, and it cannot be used to communicate directly. Neither Alice nor Bob can control the secret, rather they both contribute to it (although they

[12] For good security they want $g$ to have a large prime **order**, which is explored a little in the exercises.

can affect some of its properties in limited ways: for example if $s_a$ is chosen even then the secret must be a perfect square (mod $p$)).

Key exchange, and DHKE in particular, have become particularly important recently. If Alice and Bob set up a supposedly-secure channel (by some protocol involving public key cryptography, perhaps) then their next action could be to run the DHKE protocol and swap to the new, DHKE-derived, session key. The advantage is that an attacker who has recorded all their communications, and later manages to break the channel's confidentiality, has to solve a DHKE problem to get at the session. Furthermore, they have to do solve another DHKE problem for every other session. The security persists even after a breach of the key that created the channel.

### 5.6.1 Security

Why is the secret a secret from an eavesdropper? The security is related to the **Discrete Logarithm Problem** (DLP): from $m^x \pmod{n}$, given $m$ and $n$, compute $x$. This is the computational number theory problem (aside from factorization) that has been studied the most extensively, and it is widely believed to be hard as long as the modulus is large, notwithstanding some recent algorithmic advances.

If the DLP can be solved efficiently, then DHKE falls immediately: the attacker, who knows $g$ and $p$, can compute the secrets $s_a$ and $s_b$. But, just as the RSA Problem is not *necessarily* as difficult as factorizing, so DHKE could conceivably fall even if DLP does not. The **Diffie Hellman Problem** is: given $g$, $p$, $g^{s_a}$, $g^{s_b}$, determine $g^{s_a s_b}$. We only know that it is at least as easy as the DLP; nobody has proved it to be as difficult.

We should warn, however, of the significant weakness of the DHKE protocol. If run on an unauthenticated channel against an active adversary, it is subject to a notorious man-in-the-middle attack. We will not describe it in detail here, just consider that the attacker can share one secret with Alice and another with Bob, while Alice and Bob think that they shared one between themselves. DHKE must be authenticated in some way: either run through an already-authenticated channel, or the parties should digitally sign (e.g. with an RSA signature) their transmissions.

## Bibliography

Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO 1998,*

*18th Annual International Cryptology Conference*, volume 1462 of *Lecture Notes in Computer Science* (pp. 1–12). Springer. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.8543&rep=rep1&type=pdf.

Brent, R. P. (1999). Some parallel algorithms for integer factorisation. Expanded version of invited paper given at Euro-Par'99, available at http://maths-people.anu.edu.au/~brent/pd/rpb193.pdf.

Brumley, D. & Boneh, D. (2003). Remote timing attacks are practical. *Proceedings of the 12th USENIX Security Symposium*. Available at http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf.

Guan, D. J. (2003). Montgomery algorithm for modular multiplication. Lecture note available at http://guan.cse.nsysu.edu.tw/note/montg.pdf.

Knuth, D. E. (1998). *The Art of Computer Programming Volume 2 – Seminumerical Algorithms*. Addison-Wesley, 3rd edition.

Pomerance, C. (1996). A tale of two sieves. *Notices of the AMS*, 43(12), 1473–1485. Available at http://www.ams.org/notices/199612/pomerance.pdf.

Rivest, R. L. & Kaliski Jr., B. S. (2005). RSA problem. In *Encyclopedia of Cryptography and Security*. Available at http://people.csail.mit.edu/rivest/RivestKaliski-RSAProblem.pdf.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5), 1484–1509. Available at http://arxiv.org/abs/quant-ph/9508027.

# Chapter 6

# Message Authentication & Digital Signatures

---

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §2.5.5, 2.6.2, 5.2.2, 5.7, 6.3.6.3, 26.5-26.7 |
| **Alternatives & further depth:** | Gollmann §14.1.1, 14.3.3, 14.4 (very sketchy)<br>Ferguson et al. §2.2, 2.4, chapter 6, §7.2, 12.4.1, 12.7<br>Menezes et al. §9.2.2, 9.5, 11.1-11.2, 11.3.1-11.3.3, 11.3.6 |

---

Having discussed confidentiality and integrity, we now turn to authenticity and non-repudiation. This chapter covers two topics: MACs, which provide integrity and a limited form of authenticity, and digital signatures which provide integrity, authenticity, and non-repudiation. To do this we combine material from the previous three chapters.

Before beginning, we will revise our assumptions about the security of the communications network, to make it more realistic.

## 6.1 The Dolev-Yao Model

In security, one should always be conservative: it never hurts to overestimate the opponent, but it can be disastrous to underestimate them. So let us consider an appropriate model for communications over the internet, at the network layer.

Recall the **Internet Protocol** (IP): internet packets are transmitted from router to router until they reach their destination. Each packet has addressing information – the sender, the recipient, some timing and administrative data – and then the contents. All of these are stored in plaintext: the packet contents may have been encrypted at a higher layer (e.g. by the application which sent the packets) but the addressing information is always in the clear[1]. When a packet passes through a router, there is nothing to stop the router from altering any of the addresses, blocking it (though the transport layer protocol which sent the packet may eventually detect this), saving it and repeating it, or changing the contents.

Because there is no physical security on the internet, and even most internal networks cannot guarantee the absolute security of their cables, an enemy might perform these actions at any point of the network. The conservative assumption must be that the enemy could be anywhere and everywhere, doing anything to any packet. Thus a threat model was proposed by Dolev and Yao (Dolev & Yao, 1983), and has come to widespread acceptance for cryptographic protocols. In the model, the enemy:

- can eavesdrop on every message that passes through the network;
- can intercept any message, masquerading as the intended receiver;
- can impersonate any principal (user, agent) by sending messages in their name;
- is also a legitimate user of the network, and can send messages under their own name.

This **Dolev-Yao model** can be summarised simply: assume that the enemy carries the messages.

Do not despair. The enemy is still limited by computational feasibility, and one usually assumes that they are not allowed to block every single message thus bringing the network down. With the prior distribution of just a few keys it is possible (see chapter 7) to provide authenticated communication over such a network. We also assume that the agents themselves, the computers on the network, are not compromised by the enemy, so that their own computation and key storage is secure. (Online banks, however, are concerned with the possibility of compromised web browsers and keyloggers: this leads to some new protocols, and is a topic of current research.)

[1] This is not the case for **IPsec**, which is an alternative to the IP protocol designed to guarantee integrity of the addressing information and authenticity of the origination. IPsec is now supported by most operating systems and is widely used with VPNs. However, unsecured IP traffic is far more common.

## 6.2 Authenticity and Integrity: MACs

First, we should clarify the definition of **authenticity** in the context of the Dolev-Yao model. If Bob receives a message purporting to come from Alice, authenticity does not guarantee that he received it directly from Alice, only that Alice was the originator of the message. (This implicitly requires **integrity**, so that Bob receives Alice's message, not a tampered version.) In the Dolev-Yao model, no actor can ever be certain who their immediate correspondent is, because the attacker can always interpose themselves. It is sufficient to know that the message has arrived somehow and that Alice originated it.

We saw, in section 4.5, that a cryptographic hash can form the basis for checks on integrity, and this can be extended to authenticity (in a slightly limited way) if it becomes a **keyed hash**, which would be a function

$$h : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$$

written $h_k(x)$, with all the properties of a cryptographic hash function for each fixed $k$, plus the property that it is difficult to find $k$ and $x$ such that $h_k(x) = y$.

Rather than pursue keyed hashes, we will generalise a bit to the **Message Authentication Code** (MAC)[2]. A MAC is a function

$$MAC : \mathcal{K} \times \{0,1\}^* \to \mathcal{D},$$

where $\mathcal{K}$ is a **secret keyspace** and $\mathcal{D}$ a set of possible **digests** (usually bit streams, but not always). It comes with a corresponding **MAC verifier**

$$VER : \mathcal{K} \times \{0,1\}^* \times \mathcal{D} \to \{\text{True}, \text{False}\}.$$

For a message $m$, and given a shared secret key $k$, the purpose of the MAC is to compute a relatively short digest $MAC_k(m)$ which would detect any changes to either $k$ or $m$. The verifier will say whether any changes occurred, that is to say

$$VER_k(m', MAC_{k'}(m)) = \begin{cases} \text{True}, & \text{if } m = m' \text{ and } k = k', \\ \text{False}, \text{ (almost certainly)}, & \text{otherwise.} \end{cases}$$

[2]Things get a bit confusing here. Some literature says that keyed hashes and MACs are precisely the same thing. As far as the lecturer is concerned, the only difference is that keyed hash functions output bit streams, whereas MACs can output any type of data. Some call (unkeyed) hashes MDCs, with keyed hashes MACs, and some literature uses MIC instead of MAC. Some use "HMAC" to mean any MAC based on a hash function, but HMAC is also the name of a specific MAC. The MAC verifier is absent from some literature, with the verification forced to be $VER_k(m, d) = (d == MAC_k(m))$. Read with care.

For attackers who do not know $k$, it should be impossible to create any valid digests. Given $m_i$ and $MAC_k(m_i)$, for some values $m_i$, but not given the secret key $k$, it should be impossible to find *any* other $m$ and $d$ such that $VER_k(m, d) = \text{True}$. In particular, an attacker should not be able to change $m$ and determine the consequent change in $d$. (There are more subtle grades of security but we will return to the them in section 6.3.)

There does exist a notion of **perfect security** for MACs, similar to the Shannon security for ciphers. A MAC is perfectly secure if, for all $k$, given a message $m$ and its digest $MAC_k(m)$, for any $m' \neq m$ and any $d \in \mathcal{D}$,

$$\mathrm{P}[MAC_k(m') = d] = 1/|\mathcal{D}|.$$

That is to say, even after seeing one digest (or indeed more than one), the digest of any other message is uniformly distributed. Such a condition is not achievable in practice, and formal security of MACs follows the line of getting $\nu$-close to uniform distribution, for negligible $\nu$. But one of the exercises considers a **one-time MAC** which achieves perfect security, albeit at the unrealistic cost of changing the key for every use.

Given a MAC, Alice can send a message $m$ to Bob thus:

1. $A \to B : m$
2. $A \to B : d = MAC_k(m)$
3. $B$ accepts $m$ if $VER_k(m, d) = \text{True}$, rejects $m$ otherwise.

We have written Alice's communication of $m$ and $d$ separately to stress that they could be transmitted along completely different channels.

The integrity of the message $m$ is ensured by the security of the MAC, similarly to integrity-protection using (unkeyed) hash functions. The authenticity is also ensured, but only to a limited extent. The recipient Bob knows that only he and Alice share the secret key $k$ and so only Alice could have constructed $MAC_k(m)$. However, they can only prove authenticity of the message to other parties if they reveal the secret key, and either of Alice *or* Bob could have been the message originator.

Note that the MAC does *not* attempt to provide confidentiality. The message itself is transmitted as plaintext. Of course, Alice can encrypt the message before she sends it (of which more in section 6.5) to have confidentiality as well as the other properties.

Beware that there is nothing to stop an active attacker from intercepting message-MAC pairs and then replaying them to Bob in a different order, or multiple times. (Recall that an **authenticated channel** requires security from replay or out-of-order messages, see subsection 1.3.4.) MAC authentication only proves that Alice sent the message at some point. Of course, Alice can, and probably will, include the sender, recipient, and time of sending in the message itself; then integrity of the message guarantees that duplicates and messages out of order can be detected. But this strays into the area of protocols, which are the subject of the next chapter.

### 6.2.1 MACs Based on Ciphers

We have already seen one MAC based on encryption: given an $n$-bit block cipher, an $n$-bit MAC is $MAC_k(m) = E_k(h(m))$ and $VER_k(m, y) = (E_k(h(m)) == y)$. However this uses both encryption and hashing, which is expensive, and possibly introduces new vulnerabilities.

There are MACs based solely on ciphers. Most notable is the infamous **CBC-MAC**, which tries to make use of the fact that every block of a message affects the last cipher block under CBC symmetric encryption. Let $E_k(-)$ be some symmetric encryption function and $IV$ a public initialization vector. Split the message $m$ into blocks $m_1, \ldots, m_n$ (with some sort of padding if $m_n$ does not make a full block for the cipher). The CBC-MAC is simply the last block of ciphertext if the message is encrypted using CBC. This is equivalent to:

$$\begin{aligned} \mathit{CBC\text{-}MAC}_k\langle\rangle &= IV, \\ \mathit{CBC\text{-}MAC}_k\langle m_1, \ldots, m_n\rangle &= E_k(m_n \oplus \mathit{CBC\text{-}MAC}_k\langle m_1, \ldots, m_{n-1}\rangle). \end{aligned}$$

If the input messages are of fixed size, CBC-MAC has theoretical guarantees. But for arbitrary-length messages, CBC-MAC is flawed. If given two signed messages, $m$, $m'$, $d = \mathit{CBC\text{-}MAC}_k(m)$, $d' = \mathit{CBC\text{-}MAC}_k(m')$, it is easy to create a valid digest for another message. First, split $m'$ into blocks $m' = \langle m'_1, \ldots, m'_n\rangle$. Then

$$\mathit{CBC\text{-}MAC}(m \,\|\, (m'_1 \oplus d \oplus IV) \,\|\, m'_2 \,\|\, \cdots \,\|\, m'_n) = d'.$$

Thus some forged digests are possible. It turns out to be difficult to arrange final-block padding to defeat this attack.

To some extent this can be avoided if the MAC output is *not* the complete cipher block, but only a selection of bits from it. This prevents the $m'_1 \oplus d$ step, because $d$ does not

contain all the data necessary. However, it means than an $n$-bit digest still requires less work than $2^n$ to guess.

There is also a birthday attack on CBC-MAC, exploiting the algebraic connections between $\mathit{CBC\text{-}MAC}(m)$ and $\mathit{CBC\text{-}MAC}(m \,\|\, m')$.

There is an elementary error with CBC-MAC, when combining integrity protection with confidentiality, which is to use the same key for the MAC as for the encryption. In the case of CBC-MAC this is obviously silly because the MAC is just the last cipher block and hence easily forgeable. Various modifications exist, but the only fully secure alternative is to use a completely different key for encryption and authentication.

These problems have been fixed in a number of different ways. A leading cipher-based MAC function is called CMAC, and it works by deriving two subkeys from the MAC key and using them to pad the last block of the message: one key is used if the block is incomplete, and the other if the block is complete. Then, like CBC-MAC, the digest is some bits from the last CBC block. This does prevent the combination attack just mentioned, and in fact there is a formal proof of its security. CMAC is now a recognised standard (Dworkin, 2005).

### 6.2.2 MACs Based on Hashes

MACs based on hashes are very popular, perhaps partly because the security requirements of a MAC are similar to those of a hash. Take any $n$-bit cryptographic hash function $h$. There is an immediately obvious $n$-bit MAC:

$$MAC_k(m) = h(m \,\|\, k).$$

However, this is rather dangerous if the hash is built by iterating a compression function, allowing the prediction of $h(m \,\|\, m_1)$ from $h(m)$. An attacker who can find a hash collision $h(m) = h(m')$ can now produce many MAC collisions, by appending any message tail and *any* key,

$$MAC_k(m \,\|\, m_1) = h(m \,\|\, m_1 \,\|\, k) = h(m' \,\|\, m_1 \,\|\, k) = MAC_k(m' \,\|\, m_1).$$

The next choice would be

$$MAC_k(m) = h(k \,\|\, m).$$

This is fairly safe, because an attacker needs to find hash collisions with the correct prefix in order to find MAC collisions with a specific key. Nonetheless, the extension property

of hashes makes researchers nervous: if a message and its corresponding digest are both disclosed, it is probably possible for an attacker to compute a digest, with that key, for any extension of that message. Even the construction

$$MAC_k(m) = h(k \,\|\, m \,\|\, k)$$

has raised some security concerns, although the chance of a realistic attack seems low.

Instead, some interesting theoretical work has tended to focus minds on another format. The **HMAC** is, in simple form,

$$HMAC_k(m) = h(k \,\|\, h(k \,\|\, m)).$$

The advantage of such a function is that there are theoretical guarantees about its security, even if the underlying hash function $h$ has is not perfectly secure (and, as we have seen, most aren't!). Accordingly to Goldwasser and Bellare, the security theorem can be summarised thus: "an attacker who can forge the HMAC function can, with the same effort, either find collisions in the hash function even when the IV is random and secret, or compute an output of the compression function even with an IV that is random"(GOLDWASSER & BELLARE, 2008). Normally, a hash IV must be public (since the algorithm is unkeyed), so a weakness assuming a random IV must be a severe attack on the underlying hash. Thus the HMAC construction is, in a sense, more secure than the underlying hash it uses.

In case the key is not a multiple of the blocksize of the compression function creating $h$, and also to ensure that the two copies of $k$ have few bits in common, the full definition is

$$HMAC_k(m) = h((c_1 \oplus k) \,\|\, h((c_2 \oplus k) \,\|\, m))$$

where $k$ is zero-padded to a multiple of the blocksize, and $c_1$ and $c_2$ repeat the binary strings 01011100 and 00110110. There is nothing too special about these strings, except for the heuristic that $c_1 \oplus k$ and $c_2 \oplus k$ are not strongly correlated.

The disadvantage of the HMAC construction is that two hashes must be computed, but the outer hash is of a short input (keylength plus hash output) so this is not a substantial additional computation burden.

### 6.2.3 Other Types of MAC

There are other ways to create MACs, including **universal one-way hash functions** which are often the fastest to compute. Sadly, there is no time for them here.

## 6.3 Authenticity and Non-Repudiation: Digital Signatures

Message authentication codes provide integrity and authentication, but only to the person who shares the secret key. They do not provide **non-repudiation**. Non-repudiation would mean that Alice cannot deny having been the originator of the message. Because MAC keys are symmetric, Bob is also able to compute the same MAC, and so Alice could claim that a message from her was actually from Bob.

The idea of a **digital signature**, with binding force somehow equivalent to a physical signature, was widely discussed during the 1980s. Admittedly, paper documents are rarely fully authenticated – photo manipulation and other forgery being rather too easy – but the main force of a paper signature is to bind the signatory to some contract. The signatory should not be able to deny having signed.

Mathematical constructions which could provide a digital signature, offering integrity, authenticity, and non-repudiation, followed soon after the publication of the RSA algorithm, but they did not become widely used until the 1990s.

The formal definition of a digital signature is similar to that of a MAC, but it involves asymmetric keys. We assume that there are sets $\mathcal{K}$, a **secret keyspace**, and $\mathcal{K}'$, a **public keyspace**, and a method of generating secret-public key pairs $\langle k_s, k_v \rangle \in \mathcal{K} \times \mathcal{K}'$. $k_s$ will be called a **signing key** and $k_v$ a **verification key**. The verification key is published. There must be a function

$$\mathit{SIGN} : \mathcal{K} \times \{0,1\}^* \to \mathcal{S},$$

where $\mathcal{S}$ is a set of possible **signatures**. It comes with a corresponding **signature verifier**

$$\mathit{VER} : \mathcal{K}' \times \{0,1\}^* \times \mathcal{S} \to \{\text{True}, \text{False}\}.$$

$\mathit{SIGN}_{k_s}(m)$ is a digest of $m$ in the same vein as MACs and hashes, with security properties that we shall discuss shortly. The verifier will expose any changes to the message, or use of the wrong key:

$$\mathit{VER}_{k_v}(m', \mathit{SIGN}_{k_s}(m)) = \begin{cases} \text{True}, & \text{if } m = m' \text{ and } \langle k_s, k_v \rangle \\ & \text{are a secret-public key pair,} \\ \text{False, (almost certainly)}, & \text{otherwise.} \end{cases}$$

This means that any person can verify a signed message, but only the sender can create the signature from the message. This is what provides the non-repudiation property: only

Alice had the secret information necessary to have signed the message, so she cannot deny having originated it.

We must define the security properties of the signature function. It should be infeasible to deduce $k_s$ from $k_v$, but there is more to say than this. As with symmetric cryptography, there are a number of classes of attacks on signatures:

- In the **Key Only Attack** (KOA), the attacker only has knowledge of the verification key $k_v$.
- In the **Known Message Attack** (KMA), the attacker knows $k_v$ and a number of message-signature pairs: $m_i$ and $SIGN_{k_s}(m_i)$, $i = 1, \ldots, n$.
- In the **Chosen Message Attack** (CMA), the attacker knows $k_v$ and can choose a certain number of messages $m_1, \ldots, m_n$ of which they are then told the signature (this must exclude the signature of anything they forge). There is also an **adaptive chosen message attack** in which they may choose later messages depending on previous signatures.

KOA is analogous to extracting the secret key from a public key of an asymmetric cipher. KMA is similar to the KPA of symmetric ciphers, and CMA similar to CPA[3].

Each attack can have a different type of success:

- A **Universal Forgery** (UF) allows the attacker to sign any message they choose.
- An **Existential Forgery** (EF) allows the attacker to sign at least one message, not necessarily of their choosing.

Some authors also distinguish the UF from a **total break** which reveals the signing key itself. Some define a **Selective Forgery** (SF), which allows the attacker to sign a set of a certain non-negligible size, but we do not find the distinction between UF and SF very helpful. Clearly, achieving UF (or SF) is a more serious attack than EF, but an EF can still be serious: the classic example occurs if the message represents a sum of money to transfer to the recipient, and using an EF the recipient can change the sum to a random 32- or 64-bit number. Even with no control of the sum, they are likely to get very rich!

Existential forgery attacks are very common (as well shall see next) and many textbook signature schemes are vulnerable to them. But the gold standard for a signature scheme

[3] In fact, though, since signatures are often obtained by an inverse of encryption, in theoretical security terms the CMA is usually related to the CCA.

to be secure is that it is proof against EF even under the adaptive CMA model. As always, this is in the context of an attacker who has reasonable computing resources[4].

There is an important difference between the secret keys used for asymmetric encryption (chapter 5) and signatures. If a decryption key is disclosed, all the secrets which have ever been hidden (using the corresponding public key) are potentially revealed. Bearing in mind that the enemy can store encrypted messages for a long time, hoping to gain access to the key later, the decryption key must be kept secret for a very long time. Confidentiality hinges on its long-term secrecy.

Disclosure of a private signing key, on the other hand, is not as severe. Signed messages which have already been received still have their integrity protected, since it is too late for the attacker to alter than, and cannot be repudiated by the sender. Any signed messages received *after* the disclosure of the signing key should be discarded, because an attacker could have changed them and/or forged a signature, but messages from the past are unaffected.

This indicates that signing keys have a different status to decryption keys, but also that the disclosure of a signing key must be detected and communicated immediately to any potential signature recipients, otherwise they will not know to disregard signatures generated afterwards. This is a very topical subject, given the recent compromise of some SSL certification authorities (see chapter 7).

## 6.4 RSA Signatures

Digital signatures are often built using one-way functions (the signature) with a trapdoor (the verification). This means that the same tools used for asymmetric ciphers are often appropriate.

One of the first digital signature schemes was based on RSA encryption, and we briefly mentioned it in the last chapter. Alice generates a secret decryption key $sk_A = \langle n, d \rangle$ and a public encryption key $pk_A = \langle n, e \rangle$. The decryption key is used for signing:

$$SIGN_{sk_A}(m) = m^d \pmod{n}$$

[4] As usual, the formal definition is that a polynomial-time attacker, with access to a signature oracle, has no more than $\nu$ probability of guessing a valid signature for any message, where $\nu$ is negligible. If you wish to study the complexity-based approach to cryptographic security, see the comprehensive notes (Goldwasser & Bellare, 2008).

and the public encryption key can be used in verification:

$$VER_{pk_A}(m, x) = (x^e \equiv m \pmod{n}).$$

Because of correctness of the RSA system, the verifier functions correctly.

However, this description of RSA signatures only allows the signature of short messages (small enough to fit within the RSA modulus). Worse, it is seriously insecure.

**Claim 6.1** The simple RSA signature is universally forgeable under a chosen message attack, or existentially forgeable under either a known message attack or even a simple key only attack. (UF under a CMA, EF under KMA, and EF under KOA.)

**Proof** The first two rely on the homomorphic property of RSA (5.4), but the other is more generic.

**UF under CMA:** The attacker picks a random $r$, coprime to $n$, and asks for the two signatures

$$SIGN(r^{-1}) = (r^{-1})^d \pmod{n} \text{ and } SIGN(mr) = (mr)^d \pmod{n}.$$

Multiplying the two signatures, they have found a signature for $m$, because

$$VER((r^{-1})^d(mr)^d, m) = VER(m^d, m) = (m^{ed} \equiv m \pmod{n}).$$

**EF under KMA:** If the attacker knows two signatures

$$SIGN(m_1) = m_1^d \pmod{n} \text{ and } SIGN(m_2) = m_2^d \pmod{n}.$$

then, as above, they can multiply them to obtain a valid signature of $m_1 m_2$.

**EF under KOA:** The attacker can pick any number $x$ and compute $x^e \pmod{n}$. Then they can sign $x^e \pmod{n}$, because

$$VER(x^e, x) = (x^e \equiv x^e \pmod{n}).$$

This works because every number is the signature of *something*. ■

To avoid the EF attacks (the last one works for many other signature schemes too), it is necessary to make the signature **recognisable** in some way (this is dual to the notion

of recognisable plaintext in the COA on symmetric encryption). A simple solution is for Alice to compute $SIGN_{sk_A}(\text{ALICE} \,\|\, m)$, and the verifier to check the prefix. If signed messages are multiplied, the result will no longer begin with ALICE and will no longer verify correctly, defeating the KMA. It also defeats the KOA because it is (somewhat) infeasible for the attacker can find $x$ such that $x^e \pmod n$ begins with ALICE: if they could, it would be a partial break of RSA encryption itself.

Whether the addition of recognisable format into the signed message affects the CMA depends on whether the oracle, in the chosen message attack framework, enforces the formatting requirement. If it does, then even the UF under CMA is defeated.

A better solution is for Alice to hash the message before signing. This makes the EF attacks useless, because even if an attacker can create a valid signature for a particular hash output $h$, they will be unable to find a message which hashes to $h$ (assuming that the hash function is preimage resistant). It also defeats both attacks exploiting the homomorphic property, because the property does not also apply to hash functions. Signing a hash also has the attractive property that the message can be arbitrary length, and only a single signature computation need be performed as long as the hash output is short enough to fit within the RSA modulus. This should be true, with RSA moduli in the region of $2^{1024}$–$2^{4096}$ and hash outputs being 128–512 bits long.

However, signing a hash opens up additional attacks if hash collisions can be found. A 2nd-preimage attack is particularly dangerous: if $h(m) = y$, and message $m$ is signed by computing some function of $y$, then the same signature verifies any other message $m'$ with $h(m') = y$.

RSA signatures do not have as many traps as RSA encryption: there is no need to worry about a dictionary attack on plaintexts from a small set of possibles, and the signing exponent is not small (it is $e$, not $d$, which is often a fixed small number) so we need not worry about low-exponent attacks. However, it is still wise to pad the hash before performing the RSA exponentiation.

In chapter 5 we saw the **PKCS#1 v1.5** padding for encryption, and much the same padding is used for signatures:

$$00\ 01\ \text{PADDING}\ 00\ \text{h(MESSAGE)}$$

where $h$ is the chosen hash function and PADDING is a suitable length to fill up the RSA modulus. The verifier discards the padding, but checks the second byte: it differs from padding for encryption, so that there cannot be any confusion between encryption and signatures (see subsection 6.5) and so that an RSA signature oracle cannot be tricked into decrypting something.

A more complex padding mechanism has been proposed, based on a construction which has provable security (in much the same sense as HMAC). The so-called RSA PSS (provably secure signature) scheme[5] first produces, from the message MESSAGE, a double hash

$$H = h(\texttt{0x00}^8 \,\|\, h(\text{MESSAGE}) \,\|\, \text{SALT})$$

where `0x00`$^8$ represents 8 zero bytes, $l$ is the length of the hash output (in bytes), and SALT is up to $B - l - 2$ pseudorandom bytes. Then

$$S = \underbrace{(\text{PADDING} \,\|\, \texttt{0x01} \,\|\, \text{SALT}) \oplus MGF(H)}_{B-l-1 \text{ bytes}} \,\|\, H \,\|\, \texttt{0xbc}$$

where PADDING is $B-l-2-|\text{SALT}|$ pseudorandom bytes and $MGF$ is essentially another hash function, which outputs $B - l - 1$ bytes. It is $S$ which is fed into the RSA signature function $S^d \pmod n$.

The reason for this convoluted construction is security: apart from the last byte (to comply with some standards) $S$ is a pseudorandom string with, assuming truly pseudorandom output from the hash and MGF, statistical properties identical to true randomness. This is the basis for the formal proof of security.

## 6.5 Signing and Encryption

Digital signatures and MACs do not provide confidentiality: they do not contain the signed/digested message in full, and it must be sent by other means. In many situations we wish transmit the message confidentially, along with a signature, but there are a few pitfalls. These issues arise with both symmetric and asymmetric integrity protection (MACs and signatures) and symmetric and asymmetric confidentiality protection (secret or public key ciphers).

[5] `http://www.ietf.org/rfc/rfc3447.txt`

First, it is sensible to use a different key for signing/MAC and encryption. In the case of MACs and symmetric encryption, there are sometimes interactions between the MAC digest and the ciphertext (most obviously if using CBC-MAC and CBC encryption). In the case of asymmetric signatures and encryption, it is because the long-term security of the two keys need not be equal; as we have said above, disclosure of a signing key only affects future communications and it would be unfortunate if the disclosure also affected confidentiality of past communications too. A similar argument says that it is not a disaster if the signing key is *lost*, since a new one can be generated, but if a decryption key is lost then so is access to all the encrypted data. Again, this is a reason for using different keys when the security requirements are different.

Using different keys for signing and encryption means that work cannot be saved by using overlap between the two algorithms (if any). Cryptographers have been searching for ways to combine encryption and authentication in one construction, but most early efforts were found to be insecure.

Once we have decided that authentication and encryption are to be done separately, there is a more difficult question about the order of these operations. In what follows, $E$ denotes encryption (either symmetric or asymmetric), and $SIGN$ the integrity protection mechanism (which could be MAC rather than signature, but we continue to use the word "sign" for it). We follow (Krawczyk, 2001) and distinguish three approaches:

- **Authenticate then Encrypt (AtE):** Sign the message, then encrypt the message and signature together:
$$A \to B : E(m \,\|\, SIGN(m)).$$
This might be called the "SSL approach", as it is used by the SSL application-layer protocol (chapter 7), once symmetric session keys have been agreed.

- **Encrypt then Authenticate (EtA):** Encrypt the message, then sign the encrypted message:
$$A \to B : E(m) \,\|\, SIGN(E(m)).$$
This might be called the "IPsec approach", as it is used by the IPsec network-layer protocol designed to replace IP.

- **Encrypt and Authenticate (E&A):** Sign the message, send separately encrypted message:
$$A \to B : E(m) \,\|\, SIGN(m).$$

This might be called the "SSH approach", as it is what the SSH protocol uses once symmetric session keys have been agreed.

The community has not come to a firm conclusion on which option should be used, as you might guess from the fact that three different mainstream protocols use three different options. But there are certainly arguments in favour of one or the other depending on the application.

On theoretical grounds, EtA has the best claim: if the component MAC and encryption are perfectly secure (against CMA and CPA, respectively), then the EtA combination is also provably secure, and the same is not true for AtE or E&A. But these are theoretical arguments which have little sway in practice, perhaps because nobody actually uses perfectly secure MACs or encryption.

E&A has the minor advantage of being able to run authentication and encryption in parallel, since there are no nested operations.

EtA has the advantage of foiling the following "attack" in the case of asymmetric signatures and encryption. Suppose that Alice uses a private signing key $sk_A$ and Bob has a public encryption key $pk_B$. If Alice uses AtE and transmits

$$A \to B : E_{pk_B}(m \,\|\, \mathit{SIGN}_{sk_A}(m))$$

then Bob can, maliciously, decrypt the message and re-encrypt using Charlie's public key $pk_C$, pretending to be Alice and sending

$$B_A \to C : E_{pk_C}(m \,\|\, \mathit{SIGN}_{sk_A}(m)).$$

Thus Bob has fooled Charlie into thinking that Alice intended to send the signed message to Charlie instead of Bob. The same attack works for E&A, but it does not seem to be possible with EtA because its authentication attests that the encryption key has not been changed.

But this is not really an attack! MACs and digital signatures provide **authenticity of origin**, and it is true that Alice did originate the message $m$. If we want authenticity of intended recipient, a full protocol must be used, presumably with Alice including the name of the intended recipient (and a timestamp, to avoid replays) in the message itself. This is the subject of the next chapter.

## 6.6 Alternative Signature Schemes

RSA is far from the only digital signature scheme. There is also a **Rabin signature**, which exactly parallels the Rabin asymmetric cipher. Others include **ElGamal** and its variants Schnorr and **DSS** (Digital Signature Standard); DSS has been ratified as a NIST standard. ElGamal-style signatures require more involved mathematics, but they carry very similar lessons: some sort recognisable padding is essential, otherwise there are existential forgeries like the EF under KOA attack on RSA signatures.

## 6.7 Concluding Remarks

We have now studied mechanisms to protect integrity, authenticity of origin, and (in the case of asymmetric schemes, i.e. digital signatures) non-repudiation. RSA signatures are relatively simple to describe, but must have proper padding to avoid existential forgeries.

One thing we have not mentioned is that it is possible to reveal the signature *before* the message. Like revealing a hash, this is a form of **commitment**: at the point of commitment (when the signature is revealed) the sender is bound to one message (unless there is a flaw in the signature or an underlying hash function) and they **open** the commitment when they reveal that message. One can imagine how Alice and Bob can use this to play a game of Rock-Paper-Scissors over a network: no cheating by waiting for your opponent's move before deciding your own!

Note that the authenticity we provide is *only* authenticity of origin. On receipt of a signed message from Alice, Bob cannot be certain that Alice intended the message for him, and the message might have been intercepted and replayed or sent out-of-order by a Dolev-Yao attacker. We have repeatedly found that, as well as the message, we should encrypt/sign the name of the sender, recipient, and some sort of timestamp. These are protocols, the subject of the next chapter.

At the end of chapter 3, we commented that symmetric encryption does not exactly solve the confidentiality problem, but replaces it with a key distribution problem. Similarly, digital signature schemes do not exactly solve integrity, authenticity, and non-repudiation problems, since they still rely on the authenticity of the public keys. Under the Dolev-Yao model, Alice might publish her public encryption and signature keys but have every publication intercepted by an attacker who inserts their own keys instead. In the final chapter we will discuss approaches to the key authenticity problem.

# Bibliography

DOLEV, D. & YAO, A. C. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 198–208. Available at http://www.cs.huji.ac.il/~dolev/pubs/dolev-yao-ieee-01056650.pdf.

DWORKIN, M. (2005). Recommendation for block cipher modes of operation: the CMAC mode for authentication. NIST Special Publication 800-38B. Available at http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf.

GOLDWASSER, S. & BELLARE, M. (2008). Lecture notes on cryptography. Available at http://cseweb.ucsd.edu/~mihir/papers/gb.html.

KRAWCZYK, H. (2001). The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Advances in Cryptology – CRYPTO 2001, 21st Annual International Cryptology Conference*, volume 2139 of *Lecture Notes in Computer Science* (pp. 310–331). Springer. Full version available at http://eprint.iacr.org/2001/045.ps.

# Chapter 7

# Security Protocols

| | |
|---|---|
| **Reading (course text):** | Kaufman et al. §9.7-9.8, chapters 11 & 19 |
| **Alternatives & further depth:** | Gollmann chapter 15, §16.1, 16.5<br>Ferguson et al. chapters 7 & 13, parts of 14, 16-19<br>Menezes et al. §10.3, 10.5, 12.1-12.3, 12.9, 13.2, 13.4, 13.6 |

So far, our uses of cryptographic primitives have generally been in isolation: a single message is sent with various security properties (confidentiality, integrity, some senses of authenticity, or non-repudiation) guaranteed. Furthermore, these primitives depend on a cryptographic key: in the case of symmetric encryption or MAC keys, the key itself must have been shared with integrity and confidentiality; in the case of asymmetric encryption or signing keys, integrity of the key is required (it must be the public key corresponding to the intended recipient's secret key; we can also call this a form of authenticity). We are now in a position to discuss **security protocols**, sequences of messages between the parties which provide security properties for the stream of communications and, in some cases, address how the keys are exchanged.

Proposed security protocols usually assume the Dolev-Yao attack model, the very powerful enemy introduced in section 6.1, and also assume that the cryptographic primitives themselves are not attackable in isolation. Thus ciphers and hashes are assumed to be perfect. In fact, the prevailing view is more pessimistic, and researchers are aware that ciphers can be attacked, particularly given lots of chosen plaintext attack data (similarly

signatures, hashes, etc). So, in addition to providing security properties in the presence of perfect ciphers, we aim to avoid giving the opponent easy routes to attack the underlying primitives.

Security protocols are difficult to design. Even apparently-simple protocols can contain serious flaws, which are sometimes not discovered until years after the protocol has been proposed and widely deployed. Sometimes the flaw arises because the protocol designer was not clear about the security guarantees they thought they were providing. Sometimes they arise because a combination of multiple runs of the protocol allows an attacker to interpose themselves into the conversations. Sometimes they arise because the protocol enables a minor weakness in the encryption function, which was thought to be unimportant, to be exploited. And, as Kaufman notes (KAUFMAN et al., 2002, end of §11.1), the combination of secure schemes can, in some cases, be insecure.

Let us begin with a simple warning example. Consider the following protocol, which allows Alice and Bob to exchange a stream of (supposedly) secure messages $\langle m_1, m_2, \ldots \rangle$.

**Protocol 1.** Don't use this!

1. $A \to B$ : `Please send public key`
2. $B \to A$ : $pk_B$
3. $A \to B$ : $E_{pk_B}(k_s)$
4. $A \to B$ : $E_{k_s}(m_1)$
5. $B \to A$ : $E_{k_s}(m_2)$
6. etc.

The notation should be obvious: message 1 is sent from Alice to Bob, and so on. $k_s$ is a symmetric **session key**, in this case generated by Alice, for their stream of communication. The encryption function in message 3 is different from the rest: it is a public key cipher, the exact type of which we need not specify if we assume that it is perfectly secure, whereas the other encryptions are symmetric.

There is a variety of notation used for protocols, but it is usually fairly simple to work out what is meant. A common shorthand for encryption is $\{m\}_{k_{AB}}$ for a message encrypted by a symmetric key and $\{m\}_{pk_A}$ for an asymmetric key. (The double subscript in $k_{AB}$ indicates that it is symmetric.) This is useful for representing nested encryptions concisely. The identical notation is sometimes used for signatures, thus $\{m\}_{k_s}$ represents the signature of $m$ with signing key $k_s$: note that this usually means the concatenation

of $m$ with its signature. Some authors have used square brackets for signatures. Hashes are usually just written as $h(-)$, or more specifically like MD5$(-)$, and keyed hashes as $HMAC_k(-)$. We will stick to our established notation, but be aware of these possibilities when reading security literature.

The *idea* of protocol 1 is that Alice transmits a session key, safely encrypted using Bob's public key, and then the parties rely on the symmetric cipher for their security. We have not specified exactly what the protocol was supposed to achieve, but in fact it achieves nothing: a Dolev-Yao attacker can impersonate either party and read or tamper with any message, even if the ciphers are perfect. In the exercises you are asked to list as many different weaknesses as possible, and many attacks are illustrated in the material which follows.

## 7.1 Entity-Entity Authentication

The simplest sort of entity authentication is **unilateral**: in these examples, Alice proves her identity to Bob but Bob does not prove his to Alice. This proof depends on a shared secret or known public key, which is assumed to exist at the beginning of the protocol. Logging on by password is a simple example of unilateral authentication, but as we discussed in chapter 4 it is vulnerable to eavesdroppers and replay attacks, and therefore not suitable against a Dolev-Yao attacker.

### 7.1.1 Unilateral Authentication

Secure unilateral authentication usually follows the pattern **challenge-response**. Bob sends a challenge to Alice, to which only she knows the correct answer. Because the challenge is different every time, attackers cannot replay the response. We begin with protocols which assume that Alice and Bob share the secret key $k_{AB}$, and that nobody else knows it.

**Protocol 2.** Unilateral authentication, given symmetric key

1. $A \to B$ : Alice
2. $B \to A$ : $E_{k_{AB}}(nonce)$
3. $A \to B$ : $nonce$

Here, the word "Alice" signifies some sort of identification of Alice, perhaps a username or network address. A **nonce** is a **n**umber which is used **once**. For now, we might

think of it as a *random* number, in this case chosen by Bob, but we will discuss nonces further in subsection 7.1.3. In this particular protocol it is important that the nonce be unpredictable, otherwise an attacker can use Alice's reply to guess the next nonce value, thus impersonating her on the next authentication run.

Before even considering its advantages and disadvantages, we have to be careful about what this protocol is supposed to achieve. With a Dolev-Yao attacker, *Bob cannot know that he is talking directly to Alice*, since it is always possible that his communications are being forwarded via an attacker. All Bob can establish is that Alice is alive, and his communications with her have got through correctly on this occasion.

The disadvantage of protocol 2 is that an attacker can use Alice as a decryption oracle for any message sent using $k_{AB}$: by intercepting her request to Bob and replying with a chosen ciphertext, they can obtain the decryption of any message (once for each time Alice tries to authenticate herself to Bob)[1]. This could be defeated in a number of ways, including giving structure to the nonce (e.g. it must begin with a particular sequence) and having Alice refuse to reply if the decryption does not have the same structure. But this reduces the number of possible nonces, forcing more frequent nonce re-use. An alternative protocol is

**Protocol 3.** Unilateral authentication, given symmetric key

1. $A \to B$ : Alice
2. $B \to A$ : *nonce*
3. $A \to B$ : $E_{k_{AB}}(nonce)$ (or $MAC_{k_{AB}}(nonce)$)

This has the advantage that it can work with non-invertible function (e.g. a MAC rather than an encryption). It does not allow the attacker to run the chosen ciphertext attack, but instead they can run a chosen plaintext attack against the key $k_{AB}$.

There are modifications of protocols 2 and 3 which avoid chosen plaintext or chosen ciphertext attacks, at the cost of more work by Alice and Bob. One appears in the exercises. Another is

[1] If Bob uses the same shared key to authenticate to Alice as she does to him, this leads to an easy attack, see subsection 7.1.2.

**Protocol 4.** Unilateral authentication, given symmetric key
1. $A \to B$ : Alice
2. $B \to A$ : $E_{k_{AB}}(nonce)$
3. $A \to B$ : $E_{k_{AB}}(nonce + 1)$

If keeping the number of messages low is a priority, it is sometimes possible to authenticate using a timestamp instead:

**Protocol 5.** Unilateral authentication using timestamp, given symmetric key
1. $A \to B$ : Alice, $E_{k_{AB}}(timestamp)$

Here only one communication is needed, and another advantage is that Bob is **stateless**: he does not need to remember what challenge he sent to Alice, and is less open to denial-of-service attacks designed to make him remember too many things in a short space of time. But it is fraught with other difficulties. Bob should accept the decrypted timestamp only if it is reasonably recent, but that still leaves a window of time in which an attacker could run a replay attack. In order to avoid an attacker replaying the message, Bob would have to remember the timestamps he recently accepted, and refuse to accept duplicates, but that negates the advantage of having a stateless receiver. Furthermore, now the clock becomes part of the security system: Bob must protect his clock against alteration (e.g. an attacker setting it back in order to run a replay) and his clock must be reasonably synchronized with Alice's. Too much **clock skew** and Alice's legitimate timestamps may be rejected as old. If Alice and Bob attempt to synchronize clocks over the network, a new attack vector is created.

However, one could argue that *all* of the preceding protocols are useless, because all they achieve is that Bob knows that Alice is alive somewhere. They do not allow Alice and Bob to communicate except using their key $k_{AB}$. And if, after authenticating, Alice and Bob continue to communicate, an attacker could swoop in and impersonate Alice to Bob from that point on. Even if the attacker does not know the key $k_{AB}$, they could replay parts of old messages.

If Alice and Bob want to continue to exchange messages $m_1, m_2, \ldots$, still assured that they are talking to the right person, they must **bind** the messages to the authentication and we say that they begin an authenticated **session**. One way is to include the nonce in every future communication.

**Protocol 6.** Unilateral authentication establishing a session
1. $A \to B$ : Alice
2. $B \to A$ : *nonce*
3. $A \to B$ : $E_{k_{AB}}(nonce)$
4. $A \to B$ : $E_{k_{AB}}(m_1 \,\|\, nonce \,\|\, \text{Alice} \,\|\, counter)$
5. $B \to A$ : $E_{k_{AB}}(m_2 \,\|\, nonce \,\|\, \text{Bob} \,\|\, counter)$
6. etc.

(Step 3 can probably be omitted.) Note the structure surrounding the messages $m_i$: in addition to the nonce binding it to the authentication, there is the sender's name and a message counter which increments on each message sent. This ensures that an attacker cannot try to cause trouble by a) replaying Alice's own messages back at her, as if they were from Bob, and b) repeating messages or changing their order. Such protection is needed for a truly authenticated channel (subsection 1.3.4). A disadvantage is that the counter is presumably a fixed-length field and will eventually overrun (a sequence of over $2^{32}$ messages is not impossible!). To make replays impossible, at the point of counter exhaustion the authentication should be restarted with a new nonce. (For a truly authenticated channel, we will need to add integrity protection too, and see the discussion in section 6.5 about the combination of encryption of authentication.)

Rather than continuing to use $k_{AB}$ for the entire session, a better option is to use the authentication opening to agree a **session key**, a new symmetric key which is used for this session only and then discarded. Using a session key means that the secret $k_{AB}$ is only used during the opening of the protocol: there is less chance of an opponent managing to break this key, for which less ciphertext has been transmitted. Indeed, some cryptographers would say that a key "wears out" if used too much, because an opponent may eventually amass enough ciphertext (perhaps along with the knowledge of some of the plaintexts) to break it. We can distinguish the **long-term secret** $k_{AB}$, on which the authentication depends, from a **short-term secret** session key, and we take extra precautions to make it difficult for the long-term secret to be broken.

In the following protocol, Alice generates a session key and transmits it at stage 3:

**Protocol 7.** Unilateral authentication establishing a session

1. $A \to B$ : Alice
2. $B \to A$ : $nonce$
3. $A \to B$ : $E_{k_{AB}}(nonce \,\|\, k_s)$
4. $A \to B$ : $E_{k_s}(m_1 \,\|\, \text{Alice} \,\|\, counter)$
5. etc.

Thus there is a difference between plain **authentication** and **key establishment**: the former is only able to detect whether a user is alive, whereas the latter sets up a short-term key for an authenticated session. There are a number of different ways of doing the latter: in the example above, Alice generated the key and sent it to Bob. This is called **key transport**. One has to be careful about which party gets to generate the key because, if they are an enemy, they might choose to re-use an old key or generate a deliberately weak key (if that suits their attack). In the case above, it made sense that Alice should generate the key because it is only Alice who has proved her identity!

Rather than leave session key generation to one party, it is also possible to have both parties contribute. This is called **key agreement**. One method for this is **Diffie-Hellman Key Exchange**, which appears in the third problem sheet, and others can be found in (MENEZES et al., 1996, §12.6). Sometimes $k_s$ can be derived from exchanged nonces, but obviously not in protocol 7 where the nonce is sent in the clear.

Now let us consider authentication when one party knows the other's *public* key, under the assumption that the key is authentic. There are simple challenge-response protocols:

**Protocol 8.** Unilateral authentication, given public key

1. $A \to B$ : Alice
2. $B \to A$ : $E_{pk_A}(nonce)$
3. $A \to B$ : $nonce$

**Protocol 9.** Unilateral authentication, given public key

1. $A \to B$ : Alice
2. $B \to A$ : $nonce$
3. $A \to B$ : $SIGN_{sk_A}(nonce)$

Alice proves her identity by demonstrating her possession of the corresponding secret key. The disadvantage is that, like in protocol 2, Alice is acting as an oracle against her secret key: she will decrypt or sign anything given to her, allowing a CCA or CMA. It is absolutely vital that the same public key is not used for anything else, otherwise Alice destroys her own security by running this protocol: Alice must have a separate authentication key. And the public key cipher or signing algorithm will have to be resistant to CCA or CMA, even when almost unlimited numbers of chosen ciphertexts or messages are allowed. To some extent this can be alleviated by imposing structure on the nonce, or example requiring Bob to send $E_{pk_A}(\text{Bob} \,\|\, \mathit{nonce})$ and Alice to refuse to reply if the decrypted message is not of this form. Or there are other, longer (and therefore more expensive), protocols which avoid giving the attacker so simple a shot at CCA or CMA (one appears on the final problem sheet).

As we saw in chapter 5, asymmetric key encryption is expensive, so Alice and Bob will want to agree a symmetric session key for future communications. Simple use of this protocol contains a danger:

> **Protocol 10.** Unilateral authentication establishing a session, given public key
>
> 1. $A \to B$ : Alice
> 2. $B \to A$ : $E_{pk_A}(\mathit{nonce})$
> 3. $A \to B$ : $\mathit{nonce}$
> 4. $B \to A$ : $E_{pk_A}(\mathit{nonce} \,\|\, k_s)$
> 5. $A \to B$ : $E_{k_s}(m_1 \,\|\, \text{Alice} \,\|\, \mathit{counter})$
> 6. etc.

Bob has generated the session key, but in this unilateral authentication run he did not prove his identity to Alice. Thus it is okay for Bob to send Alice secrets using the session key, but Alice should not send secrets to "Bob" because she might actually be talking to an attacker instead. If Alice wanted to generate the session key, she would have to know Bob's public key in order to protect it:

**Protocol 11.** Unilateral authentication establishing a session, given public key

1. $A \to B$ : Alice
2. $B \to A$ : $E_{pk_A}(nonce)$
3. $A \to B$ : $E_{pk_B}(nonce \parallel k_s)$
4. $A \to B$ : $E_{k_s}(m_1 \parallel \text{Alice} \parallel counter)$
5. etc.

More likely, Alice and Bob will both contribute to the session key, perhaps using a Diffie-Hellman exchange.

### 7.1.2 Bilateral Authentication

In the preceding section, the aim was for Bob to know that he is talking to Alice. If Alice also wants to verify Bob's identity, authentication needs to be **bilateral** (also called **mutual**). It is not as simple as running two separate unilateral authentications: the attacker could impersonate Alice to Bob on one run, and Bob to Alice on the other. The two runs need to be bound together, for example by sharing a nonce. Furthermore, we might hope to make bilateral authentication more efficient than twice the work of unilateral authentication.

A simple idea, combining together the communications of two runs of protocol 3, is not secure:

**Protocol 12.** Bilateral authentication, given symmetric key: FLAWED

1. $A \to B$ : Alice, $nonce_A$
2. $B \to A$ : $E_{k_{AB}}(nonce_A)$, $nonce_B$
3. $A \to B$ : $E_{k_{AB}}(nonce_B)$

It is vulnerable to the **reflection attack**, in which the attacker intercepts all of the communications (cutting off Bob) and initiates a second run of the protocol pretending to be Bob talking to Alice. The notation $I_B$ means an attacker (**intruder**) impersonating $B$.

**Attack.** Reflection attack on protocol 12

1. $A \to I_B$ : Alice, $nonce_A$

$1'$. $I_B \to A$ : Bob, $nonce_A$

$2'$. $A \to I_B$ : $E_{k_{AB}}(nonce_A)$, $nonce_{B'}$

2. $I_B \to A$ : $E_{k_{AB}}(nonce_A)$, $nonce_B$
3. $A \to I_B$ : $E_{k_{AB}}(nonce_B)$

Bob might not even be alive, but Alice thinks he is because she was tricked into encrypting her own nonce, answering her own challenge.

If protocol 12 is extended to establish a session key, the same insecurity appears not as serious.

**Protocol 13.** Bilateral authentication establishing a session: FLAWED

1. $A \to B$ : Alice, $nonce_A$
2. $B \to A$ : $E_{k_{AB}}(nonce_A)$, $nonce_B$
3. $A \to B$ : $E_{k_{AB}}(nonce_B \parallel k_s)$
4. $A \to B$ : $E_{k_s}(m_1 \parallel \text{Alice} \parallel counter)$
5. etc.

Although an attacker can make Alice think that she is talking to Bob, they cannot get the session key so they cannot fake subsequent messages from Bob to Alice. (They can replay messages between Alice and Bob if they become a man-in-the-middle: see the following example).

The moral of this lesson is that *protocols should not be symmetrical*: Alice's and Bob's roles should not be interchangeable. An easy way to break symmetry is to put some structure in the communications or in the nonces, for example only accepting odd $nonce_A$ and even $nonce_B$. Or have the response to a nonce include the name of the sender, encrypted together with the nonce. There is also a general principle that *the initiator should be first to prove their identity*, which is not followed in protocol 12.

As in subsection 7.1.1, there are alternatives to the basic challenge-response involving timestamps or key agreement, perhaps the simplest being

**Protocol 14.** Bilateral authentication using timestamp, given symmetric key
1. $A \to B$ : Alice, $E_{k_{AB}}(\text{Alice} \,\|\, \textit{timestamp})$
2. $B \to A$ : $E_{k_{AB}}(\text{Bob} \,\|\, (\textit{timestamp} + 1))$

added to which either party can generate a session key, or they can agree on one. Care must be taken to avoid replay attacks. The same idea will appear in the mediated protocols of section 7.3.

Now let us turn to bilateral authentication based on public keys. There is an infamous protocol called **Needham-Schroeder public key protocol**[2] (NSPK) (Needham & Schroeder, 1978), which appears in various forms. The simplest is perhaps:

**Protocol 15.** Bilateral authentication, given public keys: FLAWED
1. $A \to B$ : Alice, $E_{k_B}(\text{Alice} \,\|\, \textit{nonce}_A)$
2. $B \to A$ : $E_{k_A}(\textit{nonce}_A \,\|\, \textit{nonce}_B)$
3. $A \to B$ : $E_{k_B}(\textit{nonce}_B \,\|\, k_s)$
4. $A \to B$ : $E_{k_s}(m_1 \,\|\, \text{Alice} \,\|\, \textit{counter})$
5. etc.

In the original NSPK protocol, it was suggested that the session key could be derived from the two nonces. In the version above, Alice has chosen the session key; this is an inessential difference.

The above protocol is infamous because it was proposed in 1978, had a formal proof of its security (using a logic designed to reason about actors' knowledge) published in 1989, but then showed to be flawed in 1995. The flaw is a **man-in-the-middle** attack, in which the attacker, having been contacted by Alice at some point in the past, is now able to interpose themselves between Alice and Bob with complete knowledge of the session key:

[2]Not to be confused with the Needham-Schroeder symmetric key protocol, number 17.

**Attack.** Man-in-the-middle attack on protocol 15

1. $A \rightarrow I$ : Alice, $E_{k_I}(\text{Alice} \,\|\, \mathit{nonce}_A)$

   1′. $I_A \rightarrow B$ : Alice, $E_{k_B}(\text{Alice} \,\|\, \mathit{nonce}_A)$

   2′. $B \rightarrow I_A$ : $E_{k_A}(\mathit{nonce}_A \,\|\, \mathit{nonce}_B)$

2. $I \rightarrow A$ : $E_{k_A}(\mathit{nonce}_A \,\|\, \mathit{nonce}_B)$
3. $A \rightarrow I$ : $E_{k_I}(\mathit{nonce}_B \,\|\, k_s)$

   3′. $I_A \rightarrow B$ : $E_{k_B}(\mathit{nonce}_B \,\|\, k_s)$

   4′. $I_A \rightarrow B$ : $E_{k_s}(\ldots)$

At the end of this attack, Bob thinks that he is talking to Alice (though Alice knows that she is talking to the attacker, a session which will presumably be dropped), and the attacker knows $k_s$ so that they can continue the rest of the session with Bob, impersonating Alice. Man-in-the-middle attacks are common and easy to miss. This particular attack needs Alice to have initiated an authentication run with the attacker, but this is not implausible: remember that a Dolev-Yao attacker is also a legitimate user of the network and can expect (or entice...) apparently-legitimate communication from users.

We will discuss the formal proof of protocols, in the above case obviously flawed, in section 7.6. An obvious fix for the protocol[3] (called Needham-Schroeder-Lowe) is

**Protocol 16.** Fixed version of protocol 15

1. $A \rightarrow B$ : Alice, $E_{k_B}(\text{Alice} \,\|\, \mathit{nonce}_A)$
2. $B \rightarrow A$ : $E_{k_A}(\mathit{nonce}_A \,\|\, \text{Bob} \,\|\, \mathit{nonce}_B)$
3. $A \rightarrow B$ : $E_{k_B}(\mathit{nonce}_B \,\|\, k_s)$
4. $A \rightarrow B$ : $E_{k_s}(m_1 \,\|\, \text{Alice} \,\|\, \mathit{counter})$
5. etc.

The inclusion of Bob's name in step 2 means that the message cannot be used by an attacker operating under any name other than Bob's.

### 7.1.3 On Nonces

Nonces have played a pivotal role in all the protocols we have examined, and it is possible to introduce vulnerabilities if they are used incorrectly. A nonce could be a

[3]But "obvious" is a dangerous word in protocol analysis, since researchers thought that the Needham-Schroeder protocol was "obviously" secure for 17 years before they found out that it was not.

(pseudo-)random number, a counter, or a timestamp.

Random numbers are safest in the sense that there is practically no chance that an attacker can predict them, but thanks to the birthday paradox (subsection 4.2.1) they repeat more often than counters. As long as it is 128 bits or longer, we can probably forget about the possibility of a repeat, but care must be taken to use a proper random number generator[4]. Nowadays most people would intend that a nonce should be random, but early literature on security would often allow counters and would state explicitly if randomized nonces were required. The lesson is to be certain what the nonce is used for.

It is clear why protocol 2 must use an unpredictable nonce (otherwise an enemy can predict the response and impersonate Alice). In (Ferguson et al., 2010, §11.5) the authors claim that the same is true of protocol 3 as well, but this author disagrees. Although an attacker can impersonate Alice to Bob, they can do so only in the same way that any Dolev-Yao attacker can, by interposing themselves in the message stream, and they would not recover any session key transmitted in the authentication.

Timestamps are not usually called "nonces", but they often play the same role. One must be careful about predictability, and may also rely on synchronized and secure clocks. It is common, though, for a timestamp to be included as part of the generation of a nonce, if only to supply a bit more "randomness" to the generation process.

## 7.2 Key Distribution

Authentication protocols inevitably involve a shared secret: a public or symmetric crypto key or some piece of information known to no other parties. If there are $n$ users in a network, pairwise keys require $O(n^2)$ shared secrets, which becomes infeasible for large networks. In order to reduce the requirements for so many keys, it is necessary to use **trusted third parties** (TTP).

The word **trusted** has a particular meaning in security engineering: it indicates a subject who can destroy the security of the system by behaving badly. Trusted does not mean the same as *trustworthy*, and indeed one should view trusted subjects with caution because

[4]Incidentally, the same is true for generation of session keys. A line of code incorrectly commented out, in library PRNG code in an old version of OpenSSL for Debian, led to predictable generation of weak nonces and weak session keys and caused serious flaws in a large number of servers including SSH and SSL (Appelbaum et al., 2008). It has been described as the biggest Debian bug ever.

of their power to damage the system. Nonetheless, most users can safely put their trust in some sort of authority, if only the authority which gives them physical access to their computer hardware or network connection.

Trusted third parties can help the key distribution problem by acting as a link between users, when the trusted entity shares cryptographic keys with each of them (needing $O(n)$ such keys). For example, if the University of Oxford shares authentication keys with each university member, two members who wish to authenticate can each use this trusted entity to verify the other's identity, or simply to generate a session key for them to do their own authentication.

There are two commonly-used paradigms for authentication via TTPs. One is the **key distribution centre** (KDC): if Alice wants an authenticated session with Bob, she talks securely with the TTP (using the key she shares with it), and asks it to generate a session key. Encrypted with the key Bob shares with the TTP, his copy of the session key also arrives safely (usually via Alice), and now Alice and Bob can proceed to authenticate as in the previous section.

The other method is to use the TTP to **certify** users' public keys, in which case it is called a **certification authority** (CA). In this scenario, Alice generates a public key and asks the CA to sign it with a digital signature. The CA makes whatever (physical) checks are necessary to verify Alice's identity before agreeing, and then produces this signature which is called a **certificate**. Every user has a copy of the CA's public signature verification key. Alice can then publish a signed key, and every other user has the ability to check the signature, thus preventing a Dolev-Yao attacker from interfering with the integrity of Alice's public key.

Both of these paradigms are widely used today. It is interesting to compare their relative merits. KDCs must be **online**, i.e. available whenever any two users wish to communicate. This makes them a bottleneck, and potential target for denial-of-service attack. On the other hand, a CA is only involved in the signature of a new user's key. If the CA were to be taken offline, authentication of existing users can continue uninterrupted. Furthermore, should a KDC be compromised, it can eavesdrop on every future conversation between users, whereas a compromised CA can issue false certificates and (eventually) cause false authentication between users, but cannot itself decrypt any communications because it never sees session keys. (Nonetheless, a compromised CA is a serious problem, as we mention in section 7.5).

The main disadvantage of CAs is **revocation**, which is the removal of a signed public key (due to its expiry, if a user leaves a network, or learning that it was issued in error). With KDCs it is easy to revoke a user's privileges to authenticate with other users: simply delete their key from the KDC's database. But once a public key has been signed by a CA, the signed copy can be stored and produced at any point later. Certificates do include an (integrity-protected) expiry date, but they are usually valid for a year so there needs to be another method for quicker revocation of certificates.

One solution is the **certificate revocation list**, simply a list of certificates no longer valid. It is then up to the user to check that they received a valid public key by checking that it is not on a revocation list. However, one is left with two options. Either the lists can be queried on demand, in which case the CA (or its proxy) will have to be online, defeating the advantages of the certification approach. Or the lists are generated offline and posted from time to time, in which case they will become out of date. Options such as forcing shorter expiry dates on certificates, or asking the revocation lists to be published often, still leave small windows of time in which invalid authentication may occur.

In the real world, there is no single point of trust and no single TTP. For example, users of the Oxford University network might all authenticate using a central Oxford KDC or CA, and users of the Cambridge University network via one in Cambridge. In order for Oxford and Cambridge users to authenticate across sites, a **chain of trust** must be built between them. In the case of KDCs, the Oxford KDC shares bilateral authentication keys with the Cambridge KDC and requests for authentication are transferred from one site to the other. In the case of CAs, the Oxford CA signs the Cambridge CA's verification key, and vice versa, so that an Oxford user can check the validity of a Cambridge user's public key by first checking the Oxford CA's certificate for the Cambridge CA's verification key.

More generally, a chain of trust is a path through a network such that each node trusts the next (and vice versa for bilateral authentication). Establishing such a chain can be difficult, and a common model is a tree-like structure: in simple terms, each user trusts a local TTP, local TTPs trust some regional centre, regional centres a national one, and national centres all trust each other. This is roughly how things work for CAs, with authentication possible as long as Alice and Bob both trust the same **root CAs**; validating **certificate chains** is at the heart of the authentication process in section 7.4. A network of trust authenticating public keys, along with security procedures for establishing that network, is known as **public key infrastructure** (PKI).

We will not focus on the details of PKI or the mechanisms of trust transfer, particularly with KDCs for which there are countless options, but it is a current area of interest

because some root CAs have recently shown themselves to be quite untrustworthy. We have also not addressed the question of identity and names, which are important in building a public key infrastructure.

## 7.3 Mediated Authentication

Finally, we can put together the authentication protocols with PKI to make remote authentication possible. If Alice and Bob want bilateral authentication and both trust the same root CA, they can exchange public keys along with (chains of) certificates proving their authenticity, then engage in something along the lines of protocol 16. There is nothing particularly special about the protocols involved (except for remembering to check against revocation lists).

Alternatively, they can involve a KDC, which can be trusted to generate a symmetric key $k_{AB}$ for them to use in protocol 14 or suchlike; in fact the same key can be used as a session key, since it is generated anew by the KDC each time. In practice, only the initiator communicates with the KDC (it being difficult to synchronize three-way communications over a network), and they receive a **ticket**, encrypted under Bob's key, which they forward to Bob. One such protocol is

**Protocol 17.** Needham-Schroeder with KDC

1. $A \to KDC$ : Alice, Bob, $nonce_{A'}$
2. $KDC \to A$ : $E_{k_A}(nonce_{A'} \parallel \text{Bob} \parallel k_{AB} \parallel ticket)$,
   where $ticket = E_{k_B}(\text{Alice} \parallel k_{AB})$
3. $A \to B$ : $ticket,\ E_{k_{AB}}(nonce_A)$
4. $B \to A$ : $E_{k_{AB}}(nonce_A - 1 \parallel nonce_B)$
5. $A \to B$ : $E_{k_{AB}}(nonce_B - 1)$
6. $A \to B$ : $E_{k_{AB}}(m_1 \parallel \text{Alice} \parallel counter)$
7. etc.

The idea is as follows. $k_A$ (respectively $k_B$) is a cryptographic key – which could be symmetric or asymmetric – shared *only* between Alice (respectively Bob) and the KDC: it is certainly not a public key. Note that Alice and Bob share no secrets at the start of the protocol, only their trust in the KDC. $nonce_{A'}$ is the challenge to the KDC to prove its own identity through knowledge of $k_A$. The ticket contains all the information

Bob needs to know to authenticate with Alice, but protected by Bob's key $k_B$, and Alice forwards it to him[5].

The protocol is secure under the normal attack model (if the crypto is perfect), but it does present some vulnerabilities in practice. The message at step 4, $E_{k_{AB}}(nonce_A - 1 \,\|\, nonce_B)$, must not use ECB block mode lest the two encrypted nonces be separated: it leads to a man-in-the-middle attack.

Another potential weakness occurs if an old session key $k_{AB}$ is ever compromised by an attacker (by dictionary attack, for example). Later, the attacker can intercept message 3 and replay the old ticket which first established $k_{AB}$, fooling Bob into reusing the old session key, which is not **fresh**. Freshness of tickets is addressed by the **Kerberos** protocol[6], developed by MIT in the late 1980s and early 1990s. It uses timestamps – and thus requires the system time to be included into the security system – and each ticket contains an expiry date.

**Protocol 18.** Kerberos, simplified

1. $A \to KDC$ : Alice, Bob, $nonce'_A$
2. $KDC \to A$ : $E_{k_A}(nonce'_A \,\|\, \text{Bob} \,\|\, k_{AB} \,\|\, expiry \,\|\, ticket)$,
   where $ticket = E_{k_B}(\text{Alice} \,\|\, k_{AB} \,\|\, expiry)$
3. $A \to B$ : $ticket,\ E_{k_{AB}}(timestamp)$
4. $B \to A$ : $E_{k_{AB}}(timestamp + 1)$
5. etc.

There are a number of versions of Kerberos, and the above protocol is a simplification. In the full protocol, there is a distinction between the authentication server and the ticket granting server. If permission to authenticate is revoked, there can be a time of check to time of use problem.

The Oxford "Single Sign-On" (SSO) uses the Webauth[7] system, part of which uses version 5 of Kerberos. Webauth uses SSL/TLS (see next section) for Alice's authentication of

[5] Apparently encryption under $k_B$ is used to protect the *integrity* of the ticket: we know from chapter 3 that this is a bad idea. The encryption does give Bob an assurance that the KDC has been involved in the generation of the ticket, which might help foil denial-of-service attacks.

[6] `http://www.ietf.org/rfc/rfc4120.txt`

[7] `http://webauth.stanford.edu/protocol.html`

the KDC, the KDC's authentication of Alice by a straight password, and various communications (specifically between service providers and the KDC) use Kerberos-established session keys with tickets encoded into cookies for web browsers.

In protocols using tickets, the client machines must take care to protect the confidentiality and integrity of the ticket itself, and the operating system security must be reliable if the ticket is stored in memory or on disk. When tickets are encoded as cookies, the cookies themselves must be protected, and this presents more attack possibilities: cookies have a long history of causing security flaws because they can sometimes be read or changed by malicious websites.

One other slight drawback with Kerberos, at least in the standard version, is that Alice does not authenticate herself to the KDC. Thus an attacker can request any number of bogus tickets. They will not be much help in themselves, but could be used to mount a cryptanalytic attack on the cipher, or to run a denial-of-service attack against the KDC.

Although Kerberos guarantees fresh session keys (fairly fresh, since a certain amount of clock skew must be allowed and the expiry cannot be too short), it does not give **perfect forward secrecy**. Such a level of security would mean that previous sessions are still secure from eavesdroppers *even if the long-term keys are later compromised*. It is so-named because present secrecy (or security), provided by the session key, is carried forward into the future. Clearly, this is impossible for protocols which transport the session key under the security of a long-term encryption key, but it is possible for protocols where the session key is never sent but instead agreed between the parties. A classic example is to use a Diffie-Hellman exchange to agree a session key, or use some other one-time **ephemeral key**.

## 7.4 The SSL/TLS Protocol

Rather than rely on centralised KDCs, the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are designed to run inside applications (strictly speaking in the application layer of the OSI model, or between it and the transport layer) to provide authentication via certificates, plus confidentiality and integrity of a session. SSL was designed with web browsers in mind, by Netscape in the early 1990s, but early versions contained serious flaws. TLS is its successor, introduced by the IETF in 1999[8]: TLS version 1.0 is almost, but not quite, compatible with the last version of SSL, and

[8] `http://www.ietf.org/rfc/rfc2246.txt`

is now supported by all major browsers under the **HTTPS** URI. There are some more recent revisions to the TLS protocol, bringing it to version 1.2 in 2008, but this was not deployed until recently: it was only in 2013 that the major browsers implemented TLS above version 1.0, despite major weaknesses being discovered in 2011. And all major browsers currently (November 2014) still support fallback to the last version of SSL, despite even more glaring weaknesses found there.

There are two varieties of TLS: for unilateral and bilateral authentication. Because most clients do not possess signed certificates proving their identity (they are web browsers used by people, not servers belonging to businesses) the unilateral variety is most common and is the one we will consider here. The principle is not that difficult: a client contacts a server, they agree on which cryptographic primitives to use, the server sends a certificate chain to the client, and they exchange nonces from which session keys are created. The details can be a bit ugly so we present a highly abstracted version of the protocol:

**Protocol 19.** Most basic form of TLS

1. $C \to S$ : $nonce_C$,
   Available Cipher Suites
2. $S \to C$ : $nonce_S$,
   Selected Cipher Suite,
   $\langle \text{S},\ pk_S,\ \text{ISSUER1},\ expiry,\ SIGN_{sk_{\text{Iss1}}}(\text{S} \,\|\, pk_S \,\|\, expiry) \rangle$,
   $\langle \text{ISSUER1},\ pk_{\text{ISS1}},\ \text{ISSUER2},\ expiry,\ SIGN_{sk_{\text{Iss2}}}(\text{ISSUER1} \,\|\, pk_{\text{ISS1}} \,\|\, expiry) \rangle$,
   ...
3. $C \to S$ : $E_{pk_S}(\text{Protocol Version} \,\|\, nonce_P)$
4. $C \to S$ : $E_{ke_C}\big(\text{Messages}_{1-3} \,\|\, MAC_{km_C}(\text{Messages}_{1-3})\big)$
5. $S \to C$ : $E_{ke_S}\big(\text{Messages}_{1-4} \,\|\, MAC_{km_S}(\text{Messages}_{1-4})\big)$
6. $C \to S$ : $E_{ke_C}\big(m_1 \,\|\, MAC_{km_C}(counter \,\|\, m_1)\big)$
7. $S \to C$ : $E_{ke_S}\big(m_2 \,\|\, MAC_{km_S}(counter \,\|\, m_2)\big)$
8. etc.

In the bilateral variety of TLS, the client sends their own certificate chain at step 3. There are also other parts of the protocol we have omitted, for example a request to change cipher suite, or to save and resume a session at a later date.

Here are some other comments and observations on the simplified protocol above.

1. $nonce_C$ is 32 bytes: four bytes making up the current time, and 28 random. The inclusion of the time may help preserve freshness even if the random number generator is broken (though this seems a bit futile!). A **cipher suite** includes an asymmetric cipher (RSA, ECC, or use of Diffie-Hellman key exchange instead, etc), a symmetric cipher (NULL for no cipher, the stream cipher RC4, Triple DES with CBC, AES with CBC, etc) and hash (MD5 or SHA-1). The client lists the cipher suites acceptable to them.
2. $nonce_S$ has the same properties as $nonce_C$. The protocol fails if the server doesn't like any of the client's suggested cipher suites. The certificate chain should end with a root CA trusted by the client, otherwise they will reject the authenticity of the server. In practice, web browsers come with about 300 trusted root CAs built-in (occasionally updated, but perhaps not often enough) and the user is presented with a warning if the certificate chain is invalid or leads to an untrusted root. Sadly, many users proceed despite such warnings...
3. $E_{pk_S}$ indicates asymmetric encryption using the cipher selected in message 2. The 48-byte random nonce $nonce_P$ is called the **pre-master secret**. Both client and server can now compute the 48-byte **master secret**,

$$\textsc{prf}(nonce_P, \text{“master secret”} \parallel nonce_C \parallel nonce_S),$$

where PRF is a ludicrously complex pseudorandom function (based on a combination of MD5 and SHA-1 hashes and the HMAC construction) and “master secret” is a fixed ASCII-encoded salt. Using another ludicrously complex pseudorandom function, six keys are created from the master secret: session encryption keys for messages from $C$ and $S$ ($ke_C$ and $ke_S$), IVs for these encryptions if in block mode[9], and MAC integrity-check keys for messages from $C$ and $S$ ($km_C$ and $km_S$). Note that the cryptographic keys for messages from the client to server are different from those from server to client, preventing any reflection-type attack.

4/5. The client and server prove their authenticity by returning an encrypted and integrity-protected copy of previous messages. The server can only do this if they can compute the master secret, which they can only do if they can decrypt $nonce_P$, which they can only do if they possess $sk_S$. And the certificate chain ensures that only the true server has asked for the corresponding $pk_S$ to be signed by a trusted CA.

6. The authenticated, encrypted, integrity-protected communications make use of the client and server encryption and MAC keys. The MAC uses the HMAC construc-

[9]The use of secret IVs is unusual, see section 7.5.

tion, along with the basic hash selected in the cipher suite. Note the use of the Authenticate then Encrypt paradigm (section 6.5), with the MAC wrapped inside the encryption. The message counter is included in the MAC, but not actually transmitted: this is because the recipient can keep count for themselves.

In many ways, this protocol is over-engineered. There is an unnecessary amount of hashing together of secrets and nonces in order to generate the session keys, and some of the communication is superfluous (for example the cumbersome way that the parties prove knowledge of the premaster secret in steps 4/5). The pseudorandom function has no theoretical security guarantees, although its similarities with HMAC are grounds for reassurance.

TLS is used for many vital secure applications over the internet, including email and banking. Thus there is a big prize for a malicious attacker who can break the confidentiality of the communications, or interpose themselves as a man-in-the-middle. As a result, it has been the subject of much scrutiny resulting in many attacks, most of them on specific implementations (timing attacks or exploiting a weak PRNG) rather than on the protocol itself. The complexity of the protocol is **not** a security advantage – it makes formal analysis more difficult – but there have not yet been any attacks on the *core* TLS protocol itself, if one excludes session resumption[10] and crypto- or implementation-level attacks.

SSL/TLS is not the only way to authenticate over the internet. The **IPsec** protocol exists at the IP layer (below the transport layer rather than, as TLS/SSL, above it), and encrypts packet headers and contents. It can provide packet-level authentication through a number of means, including the Internet Key Exchange (IKE) protocol. Because it is implemented in the operating system, it is transparent to applications, but this means that the level of security is only from host to host rather than application to application.

## 7.5 Recent Attacks on SSL/TLS

The TLS protocol, and where necessary additional primitives such as digital signatures, are supposed to provide the security assurances we need in order to communicate securely in a Dolev-Yao world. Many digital infrastructures rely on it: internet banking,

[10] An attack exploiting the gaps in authentication of resumed sessions was published in 2009. An article reflecting on the attack is (Farrell, 2010).

confidential email, secure voice-over-IP, and instant messaging services. Although there have been a few attacks on early versions of the SSL protocol, modern versions of the core protocol are "probably" secure against attacks.

Nonetheless, the protocol has recently had some bad press and featured in a number of malicious attacks. They attack the tools which are added to the protocol, but have been potentially very serious. There now follows a brief exposition of some recent attacks.

### 7.5.1 Attacks on Certification Authorities

First, there have been attacks on certification authorities themselves. Sadly, the certificate system requires placing one's trust entirely in the hands of these authorities, and they have not always lived up to this responsibility. Some CAs have been hacked and, potentially, their private signing keys stolen. With these keys, anyone can create a fake certificate chain. Even without the private keys, if the CA can be spoofed into issuing certificates then TLS traffic may be compromised. In the recent "Arab spring", email and other digital communications were key to co-ordinating civil disobedience. Around that time, a compromised CA was found to have issued certificates for `*.*.com` and `*.*.org`, which can certify almost any website, and particular attacks seemed to focus on major free webmail providers. It has been reported that Iranian authorities were able to snoop on the emails of thousands of people, though the veracity of such reports is difficult to establish. What is not in doubt is that fake certificates, if issued to people in a repressive regime, will allow a man-in-the-middle attack for that regime to snoop on (and even tamper with) their `https` usage, using simple tools widely available today.

A further concern is that all CAs are based in some country, by whose laws they are controlled. Should their country's authorities demand (lawfully or not...) that they create fake certificates, it would be difficult for them to refuse. Although certificates can be revoked, the revocation lists take time to be updated, and could even be blocked by the country. And there is no central system for spotting fake certificates in the first place. This exposes the essential flaw in a CA system: there is no agility of trust. The CA system was designed in the 1990s when a single, central, fixed, trusted CA was envisaged. Recent research has proposed that trust should be both distributed and easier to alter, with multiple CAs being used, compared, and dropped if found to be unworthy[11].

[11] An entertaining talk can be found at `http://www.youtube.com/watch?v=Z7Wl2FW2TcA`.

### 7.5.2 Attacks on SSL/TLS Block Modes

In late 2011 there was a lot of press about an attack called "BEAST" on the cryptography used in TLS[12]. While its application was a bit limited, it is interesting enough to be worth discussion.

The idea behind the attack has actually been known for some time (Bard, 2006). It involves the combination of two ideas: attacking the secrecy of the IVs in TLS, and using malicious code (e.g. some javascript) to get inside a web browser running a TLS session and insert chosen plaintext into the middle of important TLS-protected messages from client to server. This allows linear-time attacks on unknown ciphertext, which you saw in the first practical.

It might seem unreasonable to grant the attacker the power to insert extra plaintext into the client's message, but when a web browser creates a TLS session it is possible for malicious javascript code to send its own communication to the server, mixed up with the authentic communication from the client[13]. The use of a randomized and secret IV, however, makes it impossible for the attacker to spot equal plaintexts, since the same plaintext will encrypt to different ciphertexts with different IVs. The "BEAST" attack was defeated in TLS version 1.1, which generates a fresh random IV for each message, but it took a long time before versions 1.1 and 1.2 were deployed in practice: communications were theoretically vulnerable for more than a year after BEAST hit the headlines.

It is difficult to know where to lay the blame for this attack: with the implementations which generated a non-secret "random" IV, with the idea of keeping IVs secret in the first place, or with the general possibility of inserting malicious plaintext into TLS sessions? As with many security bugs, it is the combination of small weaknesses which lead to the attack.

A more recent (2014) attack based on CBC block mode is called POODLE[14]. This attack only works for the older SSL protocol, which unfortunately most browsers still implement: the first move of the attacker is to tamper with the packets to make the parties fall back to SSL version 3. Then a flaw in SSL, that the block padding used in CBC block mode is not verified by the MAC, is used to conduct a linear-time recovery of secret plaintext. Only 256 SSL requests are needed for each byte decrypted.

[12] `http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/`

[13] In Bard's original paper, it was assumed that the SSL/TLS session was some sort of tunnel, such as an SSL-VPN, which made it easier for malicious software to insert communication into it.

[14] `https://www.openssl.org/~bodo/ssl-poodle.pdf`

Attacks such as BEAST and POODLE are one reason why the RC4 stream cipher is a popular choice in the cipher suite (one recent study put its prevalance at over 50% of TLS traffic). Stream ciphers need no block mode at all.

### 7.5.3 Attacks on SSL/TLS Compression

There have been many recent attacks making alternative uses of the ability to insert plaintext. One is "CRIME"[15] (2012), and in 2013 a follow-up called "BREACH", which tries to determine the value of encrypted cookies by injecting plaintext *similar to cookies* and noting the size of the ciphertext. Because most browsers and servers can compress the plaintext, or at least the headers, prior to encryption (to make it quicker to encrypt and transmit) the ciphertext gets smaller when there is more redundancy in the plaintext. When the inserted data has significant overlaps with the secret plaintext cookie the cipher stream is shorter, thus leaking information about the cookie being extracted. The attack seems to be quite practical, and the only workaround is to disable compression. The upcoming revision of TLS (version 1.3) will not support compression, but this standard has not even been finalized yet, let alone deployed.

### 7.5.4 Attacks on RC4

Unfortunately, the move towards RC4 as the symmetric cipher in the suite only allows for other exploits. Although the "biases" (non-uniformity) in the byte stream produced by RC4 have long been known, in 2013 they were refined to the point where they can be used for attacks on TLS (as well as the wireless protocol WPA that succeeded WEP).

The attack[16] requires the *same* plaintext to be sent many times, using different keys: this is plausible if one prompts a server to reset the keys (which is part of the TLS protocol) and resend the same secret. Around $2^{30}$ repetitions of the same plaintext under different (unknown) keys is sufficient to get first 256 bytes of plaintext encrypted by RC4. It is practically difficult to get a server to respond to more than $2^{20}$ or so connections per hour, so the overall time for this attack is in the scale of weeks rather than seconds, but a few bytes can be deduced more quickly and this could be sufficient to weaken a secret authentication cookie.

The authors conclude that RC4 is fundamentally flawed, and predict that stronger biases will be found in future.

---

[15] `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU`

[16] `http://www.isg.rhul.ac.uk/tls/RC4biases.pdf`

This means that many of the TLS cipher suite options are insecure. As long as both server and client are sufficiently modern to implement AES encryption (or one of the 21st century alternatives), with a counter-style block mode such as Galois Counter Mode (GCM), then communication with no known flaws can happen. But neither party can permit use of RC4 or a block cipher with CBC, and both header compression and SSL fallback must be disabled. Some of these require manual configuration in current browers.

### 7.5.5 Denial-of-Service Attacks

In 2011 there was publicity about a denial-of-service attack on TLS. It appeared to make use of the fact that the server has to do the bulk of the cryptography work first, when the client sends a request to renegotiate the session keys (an advanced feature of TLS that we have not described). The authors claim[17] that the attack even works when renegotiation is disabled. Computational inequality makes it easier for a client to overload a server, even if the client has limited bandwidth. In general, it is a good idea for a protocol to make more, or at least the first, computational demands on the initiator.

The security community does not seem particularly perturbed by this attack, though, and it may turn out to be ineffective in practice. It is frustrating that attacks like these are often not published as papers in computer security journals or scientific conferences, the authors preferring to make a "presentation" at a hacking party or conference instead. It means that the details are not always full understood or scrutinized.

## 7.6 Concluding Remarks

The world of protocols is a huge area, and a topic of current research. One topic of relevance to this chapter is **zero knowledge** authentication, in which an actor proves knowledge of some data (for example a password) without revealing to anyone – including the other authentication party – any information about that data. More precisely, they cannot be used by any other party as an oracle to reveal the data. There are also protocols for many things besides authentication, including blind auctions, anonymous communication, and voting.

As we have seen, it is difficult to get protocols right. When it is "obvious" that a protocol is correct, it might be flawed. Even standing the test of time might not be a guarantee of security, as the Needham-Schroeder public key protocol demonstrated.

[17] `http://www.thc.org/thc-ssl-dos/`

Various researchers have articulated their lessons for the design of good protocols, for example (Abadi & Needham, 1995), but collected wisdom can only go so far. A danger is that a failure of *any* of the three main aspects of a secure channel (confidentiality, integrity, authenticity) causes the whole channel to fail. For example, there is no point having confidential conversations with one's enemy. How can we satisfy ourselves that a proposed protocol really is secure?

There is a large body of research on **protocol verification**. One approach is to design logics to reason about actors' knowledge, and use formal proof (or theorem proving software) to show that unauthorized actors cannot learn secret information. The pioneering example is "BAN logic", from which came a number of different formal systems. However, the infamous NSPK protocol shows the potential problems with such an approach: NSPK was proved secure in the original BAN logic, but we now know that it is not secure. This is because the logic's intruder model assumed that all principals are honest (preventing the intruder from being one of them). Because attackers are ingenious, they sometimes come up with types of attack not envisaged by the logic designer.

Another approach is to model the entire communication system as a finite state machine, describe security properties for it (taking us back to the security models material in chapter 2), and use **model checking** to verify that running protocols cannot break security. A particularly user-friendly example is Casper[18], which compiles human-readable descriptions of protocols into the process algebra CSP, then uses the FDR model checker to prove security or find attacks.

Finally, let us return to the attack model for security protocols. The Dolev-Yao model, plus an assumption of perfect cryptography, is now seen as too weak: we have already seen how protocols are designed to protect long-term secret keys from cryptanalysis, and how block modes should be considered carefully. More seriously, we should start to question the assumption that principals' computation is secure. Contemporary hackers make frequent use of keylogging software, and install kernel-mode "rootkits" which can peek inside web browsers' memory stores. If a user accesses a website secured using TLS, their assurance of its authenticity could be misplaced when the browser itself has been compromised, or their passwords and credit card information logged directly from the keyboard. It is truly impossible for Alice and Bob to create a secure session if they cannot trust their own terminals.

In response, online banks usually make use of small secure hardware devices. Such devices

[18] `http://www.cs.ox.ac.uk/gavin.lowe/Security/Casper/`

were issued to "high-net-worth" customers some years ago, and recently extended to all customers. The device is plugged into a computer or stands alone, and its main feature is fixed firmware which cannot be hacked, and whose keypresses cannot be logged. A PIN is usually required to activate the device. Authentication to the online banking website (potentially bilaterally) is performed by a challenge-response protocol between three parties: the bank, the untrusted browser, and the trusted device. See (Mannan & van Oorschot, 2011) for a description of one such protocol.

The author's bank uses a standalone device which is unique to each customer: after being activated by a PIN, the device generates a 6-digit number (a different number is generated every second). This, along with a conventional password, authenticates the user to the bank's `https://...` website. We can deduce that the device contains a hardcoded user identifier and a clock. Thus the transmission must be something like ($D$ is the device):

1. $A \to B$ : Andrew
2. $D \to A$ : $E_{k_B}(\text{Andrew} \parallel \textit{timestamp})$
3. $A \to B$ : $E_{TLS\ session}\big(\textit{password},\ E_{k_B}(\text{Andrew} \parallel \textit{timestamp})\big)$

where $E_{k_B}$ uses a key written by the bank into their device: it could be a MAC rather than an encryption. The bank will not allow timestamps which are too old, and must not permit exhaustion attacks on the 6-digit response (in general, protocols which involve human mediation have to use short codes and then take special precautions against exhaustion). Notice that the device is not providing any additional authentication of the bank, beyond that provided by TLS (running on a potentially-compromised computer, remember, which rather ruins the value of any certificates).

When the user requests to transfer money to a new payee, the device is used in a different way, with a challenge-response involving the last four digits of the payee's account number. Clearly, this is to prevent a compromised web browser from silently rewriting the destination account to that of an accomplice. However, this looks a bit dubious: the amount itself is not included in the challenge-response protocol so it cannot be authenticated, and it should not be difficult for a determined attacker to open a few thousand bank accounts (in different banks all over the country), making it quite likely that they can redirect money to them even if they cannot alter the last four digits of the account number.

At first sight, this device appears to add relatively little to security: it prevents replay of authentication credentials, and that is about all. It also makes things a bit more difficult

for the user, who needs physical possession of the device whenever they use the online banking service, and must not lose it. Given that banks lose billions of dollars to fraud every year, one would expect that they could afford to have some security experts provide a properly-designed solution.

## Bibliography

ABADI, M. & NEEDHAM, R. (1995). Prudent engineering practice for cryptographic protocols. SRC Technical Report 125, Digital Equipment Corp. Available at `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-125.pdf`.

APPELBAUM, J., ZOVI, D. D., & NOHL, K. (2008). Crippling Crypto: The Debian OpenSSL Debacle. *The Last HOPE* talk, available at `http://www.youtube.com/watch?v=QdknzkoN_aI`.

BARD, G. V. (2006). A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. Cryptology ePrint Archive: Report 2006/136. Available at `http://eprint.iacr.org/2006/136`.

FARRELL, S. (2010). Why didn't we spot that? *IEEE Internet Computing*, 14, 84–87. Available at `http://www.tara.tcd.ie/bitstream/2262/39154/1/Why%20didn't%20we%20spot%20that.pdf`.

MANNAN, M. & VAN OORSCHOT, P. C. (2011). Leveraging personal devices for stronger password authentication from untrusted computers. *Journal of Computer Security*, 19(4), 703–750. Available at `http://people.scs.carleton.ca/~paulv/papers/mpauth-jcs-revised.pdf`.

NEEDHAM, R. M. & SCHROEDER, M. D. (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21, 993–999. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.4509&rep=rep1&type=pdf`.

# Index

$*$-property, 13
$\|$ (concatenation), 4
$\langle\rangle$ (empty sequence), 4
$\oplus$ (exclusive or, XOR), 4
$\Sigma^*$ (set of sequences), 4
$\langle x, y\rangle$ (tuples and sequences), 4
2nd-preimage resistance, 47

access control, 9
access control list, 10
access control matrix, 10
access control, discretionary, 11
access control, mandatory, 11
accountability, 8
ACL, 11
active attacker, 61
adaptive CCA, 22
adaptive CMA, 95
adaptive CPA, 22
Advanced Encryption Standard, 36
AES, 36
anonymity, 7, 8
AtE, 100
attack models, 20
attacker, 5, 19
audit, 8
authenticate then encrypt, 100, 125
authenticated channel, 91
authentication, 6, 54
authentication protocol, 111
authentication token, 58
authentication, bilateral, 113
authentication, mutual, 113
authentication, unilateral, 107
authenticity, 87, 89, 90, 105
authenticity of origin, 101
authorization, 5, 9
availability, 7

BAN logic, 130
BEAST, 127
Bell-LaPadula, 11
bind, 109
birthday paradox, 51
block size, 20
BLP, 11
brute force attack, 23
buffer overrun, 15

CA, 118
Caesar cipher, 23
capabilities, 10
Casper, 130
CBC, 39
CBC-MAC, 50, 91
CCA, 22, 112
certificate, 118
certificate chain, 119, 124
certificate revocation list, 119
certification authority, 118

CFB, 40
chain of trust, 119
challenge-response, 107
checksum, 50, 62
chosen ciphertext attack, 22, 112
chosen message attack, 95, 112
chosen plaintext attack, 22, 105
cipher, 20
cipher feedback mode, 40
cipher key, 60
cipher suite, 124
cipherblock chaining, 39
ciphertext, 20
ciphertext only attack, 21
classification, 12
clearance, 12
clock skew, 109
CMA, 95, 112
COA, 21
collision resistance, 47, 63
commitment, 8, 46, 102
compression function, 47, 49
compression property, 47
computationally secure, 30
computer security, 4
computer system, 9
concatenated hash, 49
conditional probability, 26
confidentiality, 5, 19, 90
confusion, 30
cookies, 122
coprime, 72
counter mode, 40
CPA, 22, 105
CRC, 50
cryptanalysis, 24
cryptanalyst, 19
cryptographic hash function, 47
cryptosystem, 20
CTR, 40
cycle, 53

Data Encryption Standard, 32
decryption function, 20
denial-of-service, 129
DES, 32
DHKE, 84
  Security, 85
dictionary attack, 55
differential cryptanalysis, 35
Diffie Hellman Problem, 85
Diffie-Hellman, 122
Diffie-Hellman Key Exchange, 84, 111
diffusion, 30
digest, 45, 89
digital signature, 7, 94
Digital Signature Standard, 102
Discrete Logarithm Problem, 85
distribution, 26
divides relation, 72
DLP, 85
Dolev-Yao model, 88, 105
double encryption, 37
ds-property, 12
DSS, 102

E&A, 100
eavesdropper, 19
ECB, 39
ECC, 84
ECC-RSA, 84
EF, 95
electronic codebook, 39
ElGamal, 102
elliptic curves, 84

encrypt and authenticate, 100
encrypt then authenticate, 100
encryption function, 20
enemy, 19
entropy, 28
ephemeral key, 122
EtA, 100
Euclid's Algorithm, 72, 75
Euler's theorem, 73
evil hackers, 15
exhaustion attack, 23, 53
existential forgery, 95
expansion permutation, 32

Feistel structure, 31
Fermat's Little Theorem, 73
File Transfer Protocol, 58
finalization, 64
forgotten password, 54
frequency analysis, 24
fresh, 121
FTP, 58

GCHQ, 32
GNFS, 79
grant, 10
greatest common divisor, 72
group, 16, 38

hash, 45
hash chain, 56
hash stretching, 57, 60
HMAC, 93
HTTPS, 123

identification, 54
image, 47
impracticability, 27
initialization function, 41
initialization vector, 39, 49, 93, 127
integer factorization problem, 78
integrity, 6, 40, 46, 61, 89, 90, 105
Internet Protocol, 88
intruder, 113
IP, 88
IPsec, 88, 125
IV, 39

jailbreaking, 15

KDC, 118
KDF, 60
Keccak, 68
Kerberos, 121
Kerckhoffs' Principle, 21, 60
kernel mode, 14
key, 20
key agreement, 111
key derivation function, 60
key distribution centre, 118
key establishment, 111
key exchange protocol, 84
key only attack, 95
key schedule, 33
key size, 20
key transport, 111
keyed hash, 89
keylogger, 58
keyspace, 20, 89, 94
KMA, 95
known message attack, 95
known plaintext attack, 22
KOA, 95
KPA, 22

Lamport's password scheme, 57
linear, 25

linearity, 25
long-term secret, 110

MAC, 61, 89–91
MAC verifier, 89
malicious, 5
man-in-the-middle, 115
master secret, 124
MD-compliant, 65
MD-strengthening, 65
MD4, 65
MD5, 67
meet-in-the-middle, 37
Merkle-Damgård construction, 64
message authentication code, 89
messages, 20
method of repeated squaring, 76
mode, 10
model checking, 130
modular arithmetic, 72
modulus, 72
monoalphabetic substitution cipher, 24
multi-level, 12
multiplicative inverse, 72

Needham-Schroeder
    public key protocol, 115, 129
    symmetric key protocol, 120
negligible function, 30
no read-up condition, 12
no write-down condition, 13
non-repudiation, 7, 87, 94
nonce, 107, 117
NSA, 32, 35

OFB, 39
offline attack, 55
one-time MAC, 90
one-time pad, 29
one-way function, 47
one-way hash function, 47
one-way property, 47
open, 102
output feedback mode, 39
output function, 41
owner, 11

P-box, 33
padding, 64, 81, 82, 98
partial exhaustion attack, 47
passive intruder, 19
password, 45, 53, 57, 107
password database, 55
perfect forward secrecy, 122
perfect secrecy, 25
perfect security, 25, 26, 90, 105
permissions, 10
PKCS#1 v1.5, 81, 98
PKI, 119
plaintext, 20
plausible deniability, 8
power attack, 83
pre-master secret, 124
preimage resistance, 47
prime number, 72
privacy, 5
private key, 72
privilege escalation, 15
protocol notation, 106
protocol verification, 130
public key, 71
public key infrastructure, 119
public keyspace, 94

Rabin signature, 102
rainbow table, 56

random variable, 26
RC4, 41
recognisable, 97
recognisable plaintext, 21
reduced-round, 67
reference monitor, 13
reflection attack, 113
replay attack, 58
retransmission "attack", 101
revocation, 119
revoke, 10
robustness, 40
role, 16
root, 14
root certification authority, 119
rooting, 15
round, 31
round key, 31
RSA, 73
  assumption, 78
  correctness, 75
  efficiency, 75
  homomorphic property, 81, 97
  problem, 78
  security, 78
  signatures, 80, 96

S-box, 33
salt, 56, 60
same-message attack, 81
secrecy, 5
secret key, 72
secure, 23
Secure Sockets Layer, 122
security level, 12
security model, 11
security protocols, 105
seed, 40
selective forgery, 95
self-synchronization, 40
session, 109
session key, 78, 84, 106, 110, 112
SF, 95
SHA-1, 67
SHA-2, 68
SHA-3, 68
Shannon's impracticability theorem, 27
short-term secret, 110
side channel attacks, 83
signature, 94, 118
signature verifier, 94
signing key, 94
single sign-on, 121
smart card, 58
ss-property, 12
SSL, 122
SSO, 121
state, 12, 41
stateless, 109
strong collision resistance, 47
strong passwords, 55
subject, 10
subkey, 31
superuser, 14
symmetric key cryptosystem, 20
system mode, 14

ticket, 120
time of check to time of use, 14, 121
timestamp, 109, 114, 117, 121
timing attacks, 83
TLS, 122, 127
TOCTTOU, 14, 121
token, 10
total break, 95
totient, 74

tranquility property, 13
Transport Layer Security, 122
trapdoor function, 71
triple DES, 38
triple encryption, 38
truncation attack, 62
trusted third party, 117
TTP, 117

UF, 95
universal forgery, 95
universal one-way hash function, 93
UNIX file permissions, 16
unkeyed hash function, 46, 50
update function, 41
user, 10
user mode, 13
username, 53

verification key, 94
Vernam cipher, 24, 28

weak collision resistance, 47
webauth, 121
WEP, 41

XOR, 24

zero knowledge, 129