Cryptography Study Notes

by

Steven Gordon

School of Engineering and Technology
CQUniversity Australia

This book is available as:
HTML: sandilands.info/crypto/
PDF: sandilands.info/crypto/crypto.pdf

Crypto 22.03
4 January 2022 (r1973)

Contents

List of Figures vii

List of Tables xi

Glossary xiii

I Introduction 1

1 Introduction 3

2 Cryptography Concepts and Terminology 5
- 2.1 Security Concepts 5
- 2.2 Cryptography Concepts 6
- 2.3 Cryptography Notation and Terminology 8

II Tools and Techniques 9

3 Software Tools 11
- 3.1 Linux and Ubuntu 11
  - 3.1.1 Hex and Binary Viewer: xxd 11
  - 3.1.2 Arbitrary Precision Calculator: bc 13
  - 3.1.3 Random Numbers 15
  - 3.1.4 Hash Functions 16
  - 3.1.5 Bash Scripts 16
- 3.2 OpenSSL 17
  - 3.2.1 Overview of OpenSSL 17
  - 3.2.2 Common Operations 17
  - 3.2.3 Listing Ciphers and Algorithms 18
  - 3.2.4 Random Number Generation with OpenSSL 19
- 3.3 Python 20
  - 3.3.1 Cryptography Package 20
  - 3.3.2 PyCipher Package 20

4 Statistics for Communications and Security 23
- 4.1 Binary Values 23
- 4.2 Counting 25

4.3 Permutations and Combinations . . . 26
4.4 Probability . . . 28
4.5 Collisions . . . 30

5 Number Theory 31
5.1 Divisibility and Primes . . . 31
5.2 Modular Arithmetic . . . 34
5.3 Fermat's and Euler's Theorems . . . 37
5.4 Discrete Logarithms . . . 38
5.5 Computationally Hard Problems . . . 41

III Symmetric Key Encryption 43

6 Classical Ciphers 45
6.1 Caesar Cipher . . . 45
6.1.1 Caesar Cipher Definitions and Examples . . . 45
6.1.2 Brute Force Attack on Caesar Cipher . . . 48
6.2 Monoalphabetic Ciphers . . . 50
6.2.1 Monoalphabetic Cipher Definitions and Examples . . . 50
6.2.2 Brute Force Attack on Monoalphabetic Cipher . . . 51
6.2.3 Frequency Analysis Attack on Monoalphabetic Cipher . . . 51
6.3 Playfair Cipher . . . 54
6.4 Polyalphabetic Ciphers . . . 56
6.5 Vigenère Cipher . . . 57
6.6 Vernam Cipher . . . 59
6.7 One Time Pad . . . 60
6.8 Transposition Techniques . . . 61

7 Encryption and Attacks 65
7.1 Encryption Building Blocks . . . 65
7.2 Attacks on Encryption . . . 68
7.2.1 Aims and Knowledge of the Attacker . . . 69
7.2.2 Brute Force Attacks . . . 69
7.2.3 Cryptanalysis . . . 70
7.2.4 Measuring Security . . . 72
7.3 Block Cipher Design Principles . . . 73
7.4 Stream Cipher Design Principles . . . 80
7.5 Example: Brute Force on DES . . . 81
7.6 Example: Brute Force on AES . . . 83
7.7 Example: Meet-in-the-Middle Attack . . . 85
7.8 Example: Cryptanalysis on Triple-DES and AES . . . 88

8 Data Encryption Standard 91
8.1 Overview of the Data Encryption Standard (DES) . . . 91
8.2 Simplified-DES . . . 91
8.3 Details of DES . . . 97
8.4 DES in OpenSSL . . . 101

8.4.1 DES Encryption Basics in OpenSSL . . . 102
8.4.2 Symmetric Key Encryption Padding and Modes of Operation . . 103
8.4.3 DES OpenSSL Exercises . . . 106
8.5 DES in Python . . . 107

9 Advanced Encryption Standard 109
9.1 Overview of AES . . . 109
9.2 Simplified-AES . . . 111
9.3 Simplified-AES Example . . . 115
9.4 AES in OpenSSL . . . 116
9.4.1 AES Encryption Basics in OpenSSL . . . 116
9.4.2 AES Performance Benchmarking . . . 116
9.4.3 AES OpenSSL Exercises . . . 117
9.5 AES in Python . . . 118

10 Pseudorandom Number Generators 119

11 Block Cipher Modes of Operation 121
11.1 Block Ciphers with Multiple Blocks . . . 121
11.2 Electronic Code Book . . . 122
11.3 Cipher Block Chaining Mode . . . 122
11.4 Cipher Feedback Mode . . . 124
11.5 Output Feedback Mode . . . 124
11.6 Counter Mode . . . 125
11.7 XTS-AES . . . 127

IV Public Key Cryptography 129

12 Public Key Cryptography 131
12.1 Concepts of Public Key Cryptography . . . 131

13 RSA 135
13.1 RSA Algorithm . . . 135
13.2 Analysis of RSA . . . 141
13.3 Implementations of RSA . . . 143
13.4 RSA in OpenSSL . . . 145
13.4.1 RSA Key Generation in OpenSSL . . . 145
13.4.2 RSA Signing in OpenSSL (Sender) . . . 149
13.4.3 RSA Encryption in OpenSSL (Sender) . . . 149
13.4.4 RSA Decryption in OpenSSL (Receiver) . . . 150
13.4.5 RSA Verification in OpenSSL (Receiver) . . . 150
13.4.6 RSA OpenSSL Exercises . . . 150
13.5 RSA in Python . . . 151

14 Diffie–Hellman Key Exchange **153**

14.1 Diffie–Hellman Key Exchange Algorithm 153

14.2 Analysis of DHKE 156

14.3 Man-in-the-Middle Attack on DHKE 157

14.4 Implementations of DHKE 159

14.5 Diffie–Hellman in OpenSSL 159

14.6 DHKE in Python 163

15 Elliptic Curve Cryptography **165**

15.1 Overview of Elliptic Curve Cryptography 165

15.2 Applications of Elliptic Curve Cryptography 170

15.3 Elliptic Curve Cryptography in OpenSSL 171

V Authentication **173**

16 Hash Functions and MACs **175**

16.1 Informal Overview of Hashes and MACs 175

16.2 Introduction to Hash Functions 177

16.3 Properties of Cryptographic Hash Functions 178

16.4 Introduction to Message Authentication Codes 180

17 Authentication and Data Integrity **183**

17.1 Aims of Authentication 183

17.2 Authentication with Symmetric Key Encryption 184

17.3 Authentication with Hash Functions 186

17.4 Authentication with MACs 187

17.5 Digital Signatures 189

VI Key Management **191**

18 Key Distribution and Management **193**

18.1 Recommended Key Sizes 193

19 Digital Certificates **195**

VII Advances in Cryptography **197**

20 Quantum Computing and Cryptography **199**

20.1 Quantum Computing 200

20.2 Quantum Algorithms 202

20.3 Issues in Quantum Computing 205

20.4 Quantum Cryptography 207

20.5 Cryptography in the Quantum Era 208

VIII Additional Resources 209

A Cryptography Assumptions and Principles 211
- A.1 Assumptions . . . 211
  - A.1.1 Encryption . . . 211
  - A.1.2 Knowledge of Attacker . . . 212
  - A.1.3 Authentication with Symmetric Key and MACs . . . 212
  - A.1.4 Hash Functions . . . 212
  - A.1.5 Digital Signatures . . . 212
  - A.1.6 Key Management and Random Numbers . . . 212
- A.2 Principles . . . 213

B Data Formats 215
- B.1 Common Data Formats . . . 215
  - B.1.1 English Alphabet . . . 215
  - B.1.2 Printable Keyboard Characters . . . 215
  - B.1.3 Binary Data . . . 216
  - B.1.4 ASCII . . . 216
  - B.1.5 Hexadecimal . . . 217
  - B.1.6 Base64 . . . 217
- B.2 Conversions using Linux . . . 218
- B.3 Conversions using Python . . . 218

C Organisations and People in Cryptography 223
- C.1 Organisations in Cryptography and Security . . . 223
  - C.1.1 National Institute of Standards and Technology . . . 223
  - C.1.2 International Association for Cryptologic Research . . . 223
  - C.1.3 Australian Signals Directorate . . . 223
  - C.1.4 National Security Agency . . . 223
  - C.1.5 Government Communications Headquarters . . . 223
  - C.1.6 Institute of Electrical and Electronics Engineers . . . 223
  - C.1.7 Internet Engineering Task Force . . . 223
- C.2 People in Cryptography and Security . . . 224
  - C.2.1 Diffie, Hellman and Merkle . . . 224
  - C.2.2 Rivest, Shamir and Adleman . . . 224
  - C.2.3 Alan Turing . . . 225
  - C.2.4 Claude Shannon . . . 226
  - C.2.5 Hedy Lamarr . . . 227
  - C.2.6 Phil Zimmermann . . . 227
  - C.2.7 Other People . . . 228

D Versions of this Book 229

Index 231

List of Figures

2.1 Model of Encryption for Confidentiality 7

5.1 First 300 Prime Numbers 32
5.2 Powers of Integers, modulo 7 39
5.3 Discrete Logs, modulo 7 40
5.4 Powers of Integers, modulo 17 40
5.5 Discrete Logarithms, modulo 17 41

6.1 Relative Frequency of Letters by Norvig 52
6.2 Relative Frequency of Digrams by Norvig 53
6.3 Relative Frequency of N-Grams by Norvig 53

7.1 Model of Encryption for Confidentiality 66
7.2 Symmetric Key Encryption for Confidentiality 66
7.3 Common Symmetric Key Block Ciphers 68
7.4 Block Cipher with n bit blocks 73
7.5 Simple Ideal 2-bit Block Cipher 1 74
7.6 Simple Ideal 2-bit Block Cipher 2 75
7.7 Simple Ideal 2-bit Block Cipher 2 (fixed) 76
7.8 Impact of Block Sizes for 80 bit Plaintext 77
7.9 Feistel Encryption and Decryption 79
7.10 Stream Cipher Encrypt and Decrypt 80
7.11 Paul Kocher and DeepCrack 82
7.12 COPACABANA by SciEngines, 2006 82
7.13 RIVYERA S3-5000 by SciEngines, 2013 84
7.14 Double Encryption Concept 85
7.15 Example 5-bit Block Cipher 86
7.16 Solution for Meet-in-the-Middle Attack 87
7.17 Performance for Meet-in-the-Middle Attack 87
7.18 Triple Encryption Concept 88

8.1 S-DES Key Generation and Encryption 92
8.2 S-DES Key Generation and Decryption 93
8.3 S-DES Round Function Details 93
8.4 General DES Encryption Algorithm 97
8.5 Initial Permutation Tables for DES 98
8.6 Calculation of F(R,K) 98
8.7 Permutation Tables for DES 99

8.8 Definition of DES S-Boxes 1 to 4 . . . 99
8.9 Definition of DES S-Boxes 5 to 6 . . . 100
8.10 DES Permutated Choice 1 and 2 . . . 100
8.11 DES Key Generation Schedule . . . 101
8.12 DES Schedule of Left Shifts in Key Generation . . . 101

9.1 S-AES Encryption . . . 111
9.2 S-AES Decryption . . . 112
9.3 S-AES Key Generation for Round 1 . . . 113
9.4 GF(2^4) Multiplication Table used in S-AES . . . 115

11.1 ECB Encryption . . . 122
11.2 ECB Decryption . . . 123
11.3 CBC Encryption . . . 123
11.4 CBC Decryption . . . 123
11.5 CFB Encryption . . . 124
11.6 CFB Decryption . . . 125
11.7 OFB Encryption . . . 125
11.8 OFB Decryption . . . 126
11.9 CTR Encryption . . . 126
11.10CTR Decryption . . . 126

12.1 Confidentiality with Public Key Crypto . . . 133

13.1 RSA used for Confidentiality . . . 139
13.2 RSA used for Authentication . . . 139

14.1 Diffie–Hellman Key Exchange Example . . . 156
14.2 Diffie–Hellman MITM Attack Example . . . 158

15.1 Elliptic Curve for $y^2 = x^3 - 3x + 5$ . . . 166
15.2 Addition Operation on Elliptic Curve . . . 167
15.3 Self Addition on Elliptic Curve . . . 167
15.4 P + 2P on Elliptic Curve . . . 168
15.5 NP on Elliptic Curve . . . 168

16.1 Selected Cryptographic Hash Functions . . . 178

17.1 Symmetric Encryption for Authentication . . . 184
17.2 Authentication by Hash and then Encrypt . . . 186
17.3 Authentication by Encrypting a Hash . . . 186
17.4 Authentication with Hash of a Shared Secret . . . 187
17.5 Authentication with only MACs . . . 188
17.6 Digital Signature Example . . . 190

18.1 Recommend Key Lengths from ECRYPT-CSA 2018 . . . 194

20.1 Scaling the classical number field sieve (NFS) vs. Shor's quantum algorithm for factoring . . . 204

20.2 Likelihood quantum computers significant threat to public-key cryptosystems . . . 205
20.3 Quantum error rates vs qubits and intended direction of Google Quantum Research . . . 206
20.4 Example of BB84 Quantum Key Distribution . . . 207

B.1 International Reference Alphabet, or ASCII, Table in Decimal . . . 216
B.2 International Reference Alphabet, or ASCII, Table in Binary . . . 217

C.1 Diffie, Hellman and Merkle . . . 224
C.2 Rivest, Shamir and Adleman . . . 225
C.3 Alan Turing . . . 225
C.4 Claude Shannon . . . 226
C.5 Hedy Lamarr . . . 227
C.6 Phil Zimmermann . . . 228

List of Tables

2.1 Common Symbols and Notation 8

4.1 Useful Exact and Approximate Values in Binary 24

7.1 Worst Case Brute Force Time for Different Keys 69
7.2 Cryptanalysis of Triple-DES and AES 89

20.1 Worst Case Brute Force Attempts with Classical and Quantum Algorithms 203

Glossary

AAA Authentication, Authorisation and Accounting. *Classification of common security goals and mechanisms.*

ACK Acknowledgement. *Packet or frame type usually sent upon successful receipt of data.*

ADSL Asymmetric Digital Subscriber Line. *Technology used on telephone lines to provide home Internet access.*

AES Advanced Encryption Standard. *Symmetric key cipher. Recommended for use.*

ARP Address Resolution Protocol. *Maps IP addresses to MAC addresses.*

ANSI American National Standards Institute. *Standards organisation.*

AP Access Point. *Device in wireless LAN that bridges wired and wireless segments.*

ASCII American Standard Code for Information Interchange. *Format for mapping English characters to 7 bit values.*

ASD Australian Signals Directorate. *Australian government agency responsible for signals intelligence and cyber security.*

ATM Asynchronous Transfer Mode. *Wired technology used in core and access networks.*

BGP Border Gateway Protocol. *Exterior routing protocol for exchanging information between autonomous systems.*

BOOTP Bootstrap Protocol. *Used for automatically configuring computers upon boot. Replaced by DHCP.*

BSD Berkeley Software Distribution. *The original open source variant of Unix, now a popular Linux alternative for servers.*

BSSID Basic Service Set Identifier. *Unique to a wireless LAN AP; normally the AP MAC address.*

CA Certificate Authority. *Entity for signing and issuing certificates in public key cryptographic systems.*

CBC Cipher Block Chaining. *Mode of operation used to allow symmetric block ciphers to encrypt data larger than a block size.*

CCA Chosen Ciphertext Attack. *Attack category where the attacker can select ciphertext values and learn the corresponding plaintext values.*

CFB Cipher Feedback mode. *Mode of operation used to allow symmetric block ciphers to encrypt data larger than a block size*

CIA Confidentiality, Integrity and Availability. *Three broad protections expected in many computer systems.*

CLI Command Line Interface. *User interface to a computer that involves typing text based commands.*

CPA Chosen Plaintext Attack. *Attack category where the attacker can select plaintext and obtain the corresponding ciphertext.*

CPU Central Processing Unit. *The "brains" of a computer.*

CSRF Cross Site Request Forgery. *Web application attack.*

CS Computer Science. *Field of study.*

CSS Cascading Style Sheets. *Defines formatting of content in HTML.*

CTR Counter mode. *Mode of operation used to allow symmetric block ciphers to encrypt data larger than a block size*

CTS Clear To Send. *Wireless LAN control from sent in response to RTS.*

CVS Concurrent Versions System. *Version control software.*

DES Data Encryption Standard. *Symmetric key cipher. Not recommended for use.*

DDoS Distributed Denial of Service. *DoS attack coming from many computers.*

DH Diffie-Hellman. *Public key cryptography algorithm, primarily for sharing secrets.*

DHCP Dynamic Host Configuration Protocol. *Used for automatically configuring network interfaces of computers in a LAN.*

DHKE Diffie-Hellman Key Exchange. *Public key cryptography algorithm, primarily for sharing secrets.*

DNS Domain Name System. *Maps human friendly domain names to computer readable IP addresses.*

DoS Denial of Service. *Attack on server or network the prevents normal users from access the service.*

EC Elliptic Curve. *A mathematical curve used in ECC.*

ECB Electronic Code Book. *Mode of operation used to allow symmetric block ciphers to encrypt data larger than a block size.*

ECC Elliptic Curve Cryptography. *Public key cryptography approach performing operations on an elliptic curve.*

ECDH Elliptic Curve Diffie-Hellman. *Diffie-Hellman key exchange algorithm using ECC.*

ECDSA Elliptic Curve Digital Signature Algorithm. *Digital signature algorithm using ECC.*

ECIES Elliptic Curve Integrated Encryption Scheme. *Combines ECDH for key exchange with symmetric key data encryption.*

EC-KCDSA Elliptic Curve Korean Certificated-based Digital Signature Algorithm. *Digital signature algorithm using ECC.*

ECMQV Elliptic Curve Menezes–Qu–Vanstone. *Key agreement scheme based on ECC.*

EFF Electronics Frontiers Foundation. *Organisation that promotes rights and freedoms in technology.*

ESSID Extended Service Set Identifier. *Name given to wireless LAN network; multiple APs may be in the same network.*

FPGA Field Programmable Gate Array. *An integrated circuit that can be programmed by the end user (as opposed by the manufacturer).*

FTP File Transfer Protocol. *Application layer protocol for transferring files between client and server. Uses TCP.*

FSF Free Software Foundation. *Organisation that promotes the use of free (as in freedom), open source software.*

GCHQ Government Communications Headquarters. *UK government signals intelligence agency.*

GUI Graphical User Interface. *User interface to a computer that involves windows, mouse, buttons etc.*

GNU GNU's Not Unix. *A free operating system, using free, open source software. Often combined with Linux kernel to produce GNU/Linux.*

HMAC Hash-based MAC. *Message authentication code function that uses existing hash algorithms. That is, converts hash functions into MAC functions.*

HTML HyperText Markup Language. *Language for defining how content is displayed in a web browser.*

HTTP HyperText Transfer Protocol. *Application layer protocol for transferring web pages from server to client. Uses TCP.*

HTTPS HTTP Secure. *HTTP on top of SSL/TLS, to provide secure web browsing.*

IACR International Association for Cryptologic Research. *Non-profit scientific organisation that publishes latest cryptology research in journals and conferences.*

IANA Internet Assigned Numbers Authority. *Organisation that defines the use of Internet numbers such as ports and protocol numbers.*

ICMP Internet Control Message Protocol. *Protocol for testing and diagnostics in the Internet. Used by ping.*

IDE Integrated Development Environment. *Software application used for developing, testing and debugging software.*

IEEE Institute of Electrical and Electronic Engineers. *Organisation that defines electrical, communications and computer standards, including for LANs and WLANs.*

IETF Internet Engineering Task Force. *Organisation that defines standards for Internet technologies, including IP, TCP and HTTP.*

IP Internet Protocol. *Network layer protocol used for internetworking. Core protocol of the Internet. Two versions: IPv4 and IPv6.*

IPsec Internet Protocol Security. *Extensions to IP that include security mechanisms. Optional whan using IPv4.*

ISAKMP Internet Security Association and Key Protocol. *Security protocol for key exchange.*

ISP Internet Service Provider. *Organisation that provides Internet access to customers.*

IT Information Technology. *Field of study.*

IV Initialisation Vector/Value. *Value used to initialise cryptographic algorithms. Often chosen by user similar to a key.*

KPA Known Plaintext Attack. *Attack category where the attacker knows pairs of plaintext/tciphertext.*

LAN Local Area Network. *Network covering usually offices, homes and buildings. Layer 1 and 2 technology.*

LCG Linear Congruential Generator. *Pseudo random number generator.*

LTS Long Term Support. *Assigned to selected versions of software, such as Ubuntu operating system, to indicate that version will be supported for a long period than other versions.*

MAC Message Authentication Code or Medium Access Control

MDC Modification Detection Code. *The functionality provided by unkeyed hash functions in data authentication.*

MD5 Message Digest 5 hash function. *Cryptographic hash function that is still widely used, but no longer considered secure for many purposes.*

MITM Man-in-the-Middle. *An attack where an attackerintercepts messages between two parties, masquerading as those two communicating parties.*

NAT Network Address Translation. *Technique used in networks to convert private, internal IP addresses into public, external IP addresses.*

NIC Network Interface Card. *Device in a computer that connects the computer to a network.*

NIST National Institute of Standards and Technology. *US standards organisation that includes key standards and processes in security and cryptography.*

NTP Network Time Protocol. *Protocol for clients to synchronise their clocks to more accurate time servers.*

NSA National Security Agency. *US government agency responsible for signals intelligence and cryptography.*

OFB Output Feedback mode. *Mode of operation used to allow symmetric block ciphers to encrypt data larger than a block size*

OS Operating System. *Software that provides services for operating a computer, hiding computers details from applications.*

OSI Open Systems Interconnection. *Standard for connecting different networks together. No longer widely used by the OSI 7 layer model still referred to.*

OSPF Open Shortest Path First. *Internal routing protocol.*

OTP One-Time Pad. *Unbreakable, but often impractical, cipher.*

OWASP Open Web Application Security Project. *Project that keeps track of common attacks on web applications and provides advice on securing apps.*

PAM Pluggable Authentication Modules. *Linux modules that allow application to use different authentication techniques.*

PHP PHP: Hypertext Preprocessor. *Programming language primarily used to create dynamic web sites.*

PHY Physical Layer. *Lowest layer in Internet and OSI layer architectures. Deals with transmitting bits as signals.*

PRNG Pseudo Random Number Generator. *Algorithm for outputting random numbers. Not a true random number generator, but commonly used for convenience.*

PSEC Provably Secure Elliptic Curve Encryption. *Data encryption using ECC.*

PSK Pre-Shared Key. *Secret cryptographic key that two parties have exchanged in advance.*

QKD Quantum Key Distribution. *A secret key sharing protocol based on quantum technology.*

RAM Random Access Memory. *Short term, volatile storage area for computers.*

RFC Request For Comment. *Type of standard used by IETF. The standards for IP, TCP and DNS are RFCs.*

RIP Routing Information Protocol. *Internal routing protocol.*

RSA Rivest Shamir Adleman cipher. *Public key cryptographic cipher used for confidentiality, authentication and digital signatures.*

RTS Request To Send. *Type of WLAN frame.*

RTT Round Trip Time. *Time for a message to travel from source to destination and then back to the source.*

S-AES Simplified Advanced Encryption Standard. *An educational cipher that mimics AES but using smaller values that allow calculation by hand.*

SCP Secure Copy. *Command and protocol for transferring files securely from one computer to another.*

S-DES Simplified Data Encryption Standard. *An educational cipher that mimics DES but using smaller values that allow calculation by hand.*

SDH Synchronous Digital Hierarchy. *Wide area network technology used across cities and countries.*

SHA Secure Hash Algorithm. *Cryptographic hash algorithm. Different variants including SHA, SHA2 and SHA3.*

SMTP Simple Mail Transfer Protocol. *Application layer protocol for transferring email between computers.*

SPI Stateful Packet Inspection. *Technique that allows a firewall to make decisions on packets based on past packets in a connection.*

SQL Structured Query Language. *Language for querying databases.*

SSH Secure Shell. *Application for remotely logging in to a computer.*

SSID Service Set Identifier. *Same as a ESSID.*

SSL Secure Sockets Layer. *Protocol for securing application data that uses TCP for communications. Replaced by TLS but still referred to.*

SVN Subversion. *Version control system.*

SYN Syncrhonise. *Type of TCP segment, used during connection establishment phase.*

TCP Transmission Control Protocol. *Transport layer protocol that provides reliable, connection-oriented data transfer. Used by many applications in the Internet.*

TFTP Trivial File Transfer Protocol. *Application layer protocol for transferring files. Very lightweight, compared to FTP.*

TLS Transport Layer Security. *Replaced SSL.*

Tor The Onion Router. *System for private networking, whereby it is very difficult for someone to know who you are communicating with.*

TTL Time To Live. *Value often given to packets so that after a certain time those packets are discarded/deleted. Usually measures in router hops, rather than seconds.*

UDP User Datagram Protocol. *Transport layer protocol that provides unreliable, connection-less data transfer. Used by applications that require simplicity and/or fast data transfer. Alterative to TCP.*

URL Uniform Resource Locator. *Identifies a resource in the Internet, such as a web page. E.g. http://www.example.com/dir/page.html*

VM Virtual Machine. *Software implemtnation of a computer, virtualising the typical hardware components of a computer.*

vn virtnet. *Software for quickly deploying Linux based virtual machines in a virtual network.*

VPN Virtual Private Network. *Technology for private communications from a client to server.*

W3C World Wide Web Consortium. *Organisation that sets standards for web browsing and applications, such as HTML.*

WAN Wide Area Network. *Network that covers cities and countries, usually owned by telecom operators or ISPs.*

WiFi Wireless Fidelity . *Marketing name for WLAN.*

WLAN Wireless Local Area Network. *Technology for wireless communications on a LAN.*

WPA WiFi Protected Access. *Encryption and authentication protocol for WLANs.*

WSL Windows Subsystem for Linux. *Software that allows command-line based Linux operating systems to run as an application in Windows.*

XML eXtensible Markup Language. *Language for defining other languages that define the structure/organisation of content.*

XSS Cross Site Scripting. *Web application attack.*

Part I

Introduction

Chapter 1

Introduction

This book is a collection of definitions, algorithms, examples and notes on cryptography.

This book is work-in-progress. That is, it is incomplete. Content will be added regularly, so you are encouraged to check for new versions frequently.

This book was created with LaTeX, with the PDF book, website and lecture material generated from the one set of LaTeX source files. Links to the lecture slides are included at the start of each chapter. You can also obtain the lecture slides and LaTeX source from the following directories:

- Slides for presentation, including PDF slides (`slides-colour.pdf`), PDF handouts including notes (`handout-colour.pdf`), LibreOffice Impress slides with notes (`slides-colour.odp`), Microsoft PowerPoint slides with notes (`slides-colour.pptx`) and PDF handouts in black and white for printing (`handout-print.pdf`). Note the ODP and PPTX slides only contain images of each slide, so cannot be easily edited, but can be used in dual screen presentation mode. `https://sandilands.info/crypto/slides/`
- LaTeX source for the book (including all the .tex, images and style files) as well as selected examples: `https://sandilands.info/crypto/source/`

Video
Introduction to Cryptography Study Notes (6 min; Dec 2021)
https://www.youtube.com/watch?v=s7HmFPvw8nc

File: crypto/intro.tex, r1960

Chapter 2

Cryptography Concepts and Terminology

This chapter defines key concepts and terminology in cryptography. This would likely to have been already covered in a subject that introduces computer security (e.g. an introduction to networking or IT security). Therefore it is quite brief, serving mainly a refresher and to set the scene for subsequent chapters.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

Video
Cryptography Concepts and Terminology mini-lecture (14 min; Feb 2020)
https://www.youtube.com/watch?v=fx4nGsoum6A

2.1 Security Concepts

There are three broad protections that are considered important when securing information systems:

Confidentiality ensures only authorised parties can view information

Integrity ensures information, including identity of sender, is not altered

Availability ensures information accessible to authorised parties when needed

Examples of confidentiality: a file is encrypted so that only authorised party (with a secret key) can decrypt to read the contents of the file; web traffic sent across Internet is encrypted so that intermediate users cannot see the web sites and content of web pages you are visiting.

Examples of integrity: If someone maliciously modifies a message, the receiver can detect that modification; if someone sends a message pretending to be someone else, the receiver can detect that it is a different person.

File: crypto/concepts.tex, r1961

Examples of availability: a web server provides customers ability to buy products; that web server is available for the customers 24/7 even under malicious attacks.

While Confidentiality, Integrity and Availability (CIA) is commonly referred to, it does not cover everything. There are other classifications of protections. Often Authentication, Authorisation and Accounting (AAA) is also referred to as additional protections.

Authentication ensures that the individual is who she claims to be (the authentic or genuine person) and not an impostor

Authorisation providing permission or approval to use specific technology resources

Accounting provides tracking of events

Example of authentication: check username and password when user logs into system.

Example of authorisation: check that user is authorised to access a particular document.

Example of accounting: record logs of who accesses files and provide summary reports.

This book does not attempt to cover everything. The scope is:

- Focus on confidentiality and integrity of information using technical means
- Means of authentication also covered
- Accounting, system availability, policy, etc. are out of scope
- See other subjects or books on “IT Security”, “Network Security Concepts” or similar

2.2 Cryptography Concepts

Encryption is a key mechanism in cryptography, and often used to provide confidentiality.

- Aim: assure confidential information not made available to unauthorised individuals (data confidentiality)
- How: encrypt the original data; anyone can see the encrypted data, but only authorised individuals can decrypt to see the original data
- Used for both sending data across network and storing data on a computer system

While encryption is used to provide different services in cryptography, the main service is confidentiality: keeping data secret. In the following we talk about using encryption for confidentiality. Later we will see that the same encryption mechanisms can also provide other services such as authentication, integrity and digital signatures.

Figure 2.1 shows a simple model of system that uses encryption for confidentiality. Assume two users, A and B, want to communicate confidentially. User A has a plaintext message to send to B. User A first encrypts that plaintext using a key. The output ciphertext is sent to user B (e.g. across the Internet). We assume the attacker, user C, can intercept anything sent – in this case they see the ciphertext. User B receives the

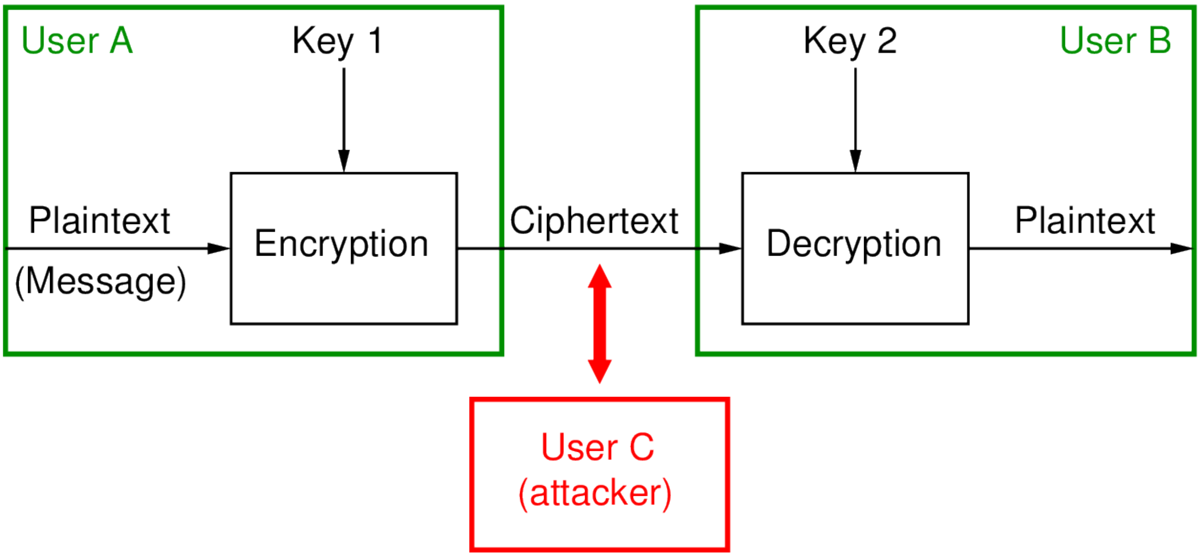


Figure 2.1: Model of Encryption for Confidentiality

ciphertext and decrypts. If the correct key and algorithm is used, then the output of the decryption is the original plaintext.

The aim of the attacker is to find the plaintext. They can either do some analysis of the ciphertext to try to discover the plaintext, or try to find the key (if the attacker knows key 2, they can decrypt the same as user B).

In symmetric key crypto, Key 1 and Key 2 are identical (symmetry of the keys).

In public key crypto, Key 1 is the public key of B and Key 2 is the private key of B. (asymmetric of the keys).

Given the above scenario, the important terms in cryptography are:

Plaintext original message

Ciphertext encrypted or coded message

Encryption convert from plaintext to ciphertext (enciphering)

Decryption restore the plaintext from ciphertext (deciphering)

Key information used in cipher known only to sender/receiver

Cipher a particular algorithm (cryptographic system)

Cryptography study of algorithms used for encryption

Cryptanalysis study of techniques for decryption without knowledge of plaintext

Cryptology areas of cryptography and cryptanalysis

2.3 Cryptography Notation and Terminology

Mathematical notation is often used when describing cryptographic mechanisms, as it is precise and brief. Table 2.1 summarises the main notation used in this book. For brevity, single letters are often used to refer to information or operations. However sometimes we need to change the letter to avoid ambiguity. For example, P often refers to plaintext. However in public key cryptography we see the letter P used in the public key (PU) and private key (PR), and so use M to refer to the plaintext (or message).

Operations, such as encryption, decryption and hashes, are written as functions, where the inputs to the operation are given as parameters within parentheses, and the output of the operation is assigned to the variable to the left of the equal sign. For encryption and decryption, the inputs are commonly order as key then data (plaintext or ciphertext).

Subscripts are often used to identify either the information belongs to a particular user or that the operation uses a specific algorithm. For example, PR_A is a private key that belongs to user A, and K_{XY} is a secret key shared between users X and Y. $H_{MD5}(M)$ means apply a hash function, specifically the MD5 algorithm.

Symbol	*Description*	*Example*
P	Plaintext or message	$P = \mathrm{D}(K_{AB}, C)$
M	Message or plaintext	$M = \mathrm{D}(PR_B, C)$
C	Ciphertext	$C = \mathrm{E}(K_{AB}, P)$ or $C = \mathrm{E}(PU_B, M)$
K	Secret key, symmetric key	
K_{AB}	Secret key shared between A and B	
E()	Encrypt operation	$\mathrm{E}(K_{AB}, P)$ or $\mathrm{E}(PU_B, M)$
$\mathrm{E}_{cipher}()$	Encrypt operation using cipher	$\mathrm{E}_{AES}(K_{AB}, P)$
D()	Decrypt operation	$\mathrm{D}(K_{AB}, C)$ or $\mathrm{D}(PR_B, C)$
PU_A	Public key of user A	
PR_A	Private key of user A	
H()	Hash operation	$\mathrm{H}(M)$
MAC()	MAC operation	$\mathrm{MAC}(K_{AB}, M)$
XOR, $\oplus$	Exclusive OR operation	A XOR B, $A \oplus B$
h	Hash value	$h = \mathrm{H}(M)$
$\|\|$	Concatenate (join) operation	$A\|\|B$

Table 2.1: Common Symbols and Notation

Part II

Tools and Techniques

Chapter 3

Software Tools

This chapter lists common tools referred to within the rest of the book. The purpose is to make you aware of the tools; not to teach you how to use the tools. While some setup and basic usage instructions may be given, you can normally find detailed instructions by searching online, or within the tool help or manual pages.

3.1 Linux and Ubuntu

Almost all of the software-based examples or demonstrations in this book are performed using a Linux operating system, especially using command-line tools (as opposed to tools using a graphical user interface).

This book uses the Ubuntu distribution of Linux, although most tools will work equally well on other distributions (e.g. Red Hat, Fedora, Debian, Slackware).

It is assumed you have access to a Linux command line and having basic knowledge of common file and directory operations in Linux. My other book Network and Security in Linux motivates the use of Linux in the field (Chapter 2) and provides an introduction to the command line (Chapter 4).

The following lists selected command line tools which are normally available in common Linux distributions.

3.1.1 Hex and Binary Viewer: xxd

We often deal with binary values in cryptography. Therefore it is useful to be able to view the contents of binary files (as opposed to text files). `xxd` is one tool that allows viewing any file as binary or hex (default).

To demonstrate, let's first create a plaintext file called `demo.txt`:

```
$ echo -n "This is a super secret message. " > demo.txt
$ ls -l demo.txt
-rw-r--r-- 1 sgordon sgordon 32 Dec 5 17:58 demo.txt
```

While your output of the `ls` command may be different it should show 32, meaning the file size is 32 bytes (note the text contains 32 characters, including the space at the end; the `-n` option means no new line character is added).

File: crypto/tools.tex, r1962

We can view the text file with cat:

```
$ cat demo.txt
This is a super secret message. $
```

The command prompt does not start on a new line since our text file does not finish with a new line character.

Now let's look at the file in hex and then binary format (using the -b option):

```
$ xxd demo.txt
00000000: 5468 6973 2069 7320 6120 7375 7065 7220 This is a super
00000010: 7365 6372 6574 206d 6573 7361 6765 2e20 secret message.
$ xxd -b demo.txt
00000000: 01010100 01101000 01101001 01110011 00100000 01101001 This i
00000006: 01110011 00100000 01100001 00100000 01110011 01110101 s a su
0000000c: 01110000 01100101 01110010 00100000 01110011 01100101 per se
00000012: 01100011 01110010 01100101 01110100 00100000 01101101 cret m
00000018: 01100101 01110011 01110011 01100001 01100111 01100101 essage
0000001e: 00101110 00100000                                     .
```

By default, xxd shows three pieces of information per line of output:

1. The address, in hex, of first byte on the line
2. The hex (or binary) values, with some spacing to ease the readability
3. The ASCII character (if it is printable) for the corresponding byte. A dot is shown if the ASCII character is not printable (e.g. the DELete or ESCape characters).

xxd has a variety of command line options, which are well described in the man page. For example, to show 8 bytes per line (column size), group into 2 sets of 4 bytes, displaying a length just the first 16 bytes:

```
$ xxd -c 8 -g 4 -l 16 demo.txt
00000000: 54686973 20697320 This is
00000008: 61207375 70657220 a super
```

Combined with other commands, such as cut and grep, you can extract information of interest. For example, show the binary representation of the first 64 bytes of the file /bin/ls, 8 bytes per line:

```
$ xxd -b -l 64 -c 8 -g 8 /bin/ls | cut -d " " -f 2
0111111101000101010011000100011000000010000000010000000100000000
0000000000000000000000000000000000000000000000000000000000000000
0000001100000000001111100000000000000001000000000000000000000000
0101000001011000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
1010000000000011000000100000000000000000000000000000000000000000
0000000000000000000000000000000001000000000000000011100000000000
0000100100000000010000000000000000011100000000000001101100000000
```

3.1.2 Arbitrary Precision Calculator: bc

No doubt you have access to a software calculator on your computer or phone, or even a traditional calculator on your desk. While these calculators are convenient, they often make approximations when dealing with very large numbers. An arbitrary precision calculator will give exact answers, no matter how big numbers are. Several important ciphers in cryptography rely on calculations with big numbers, so it is nice to have an arbitrary precision calculator available. In Linux, `bc` is a command-line arbitrary precision calculator.

The following shows an example of starting `bc` in Linux, and then performing normal arithmetic operations.

```
$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free
    Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
<kbd>1␣+␣2</kbd>
3
<kbd>20␣-␣13</kbd>
7
<kbd>7␣*␣6␣+␣2</kbd>
44
<kbd>7␣*␣(6␣+␣2)</kbd>
56
<kbd>56␣/␣8</kbd>
7
<kbd>54␣/␣10</kbd>
5
```

Note that the last division gives the quotient as the answer, not a fraction. By default, fractions are not used, but can easily be enabled by setting the *scale* parameter as follows:

```
scale=2
54/10
5.40
```

Exponentiation is also supported using the “hat” or “carat” operator:

```
10^2
100
2^3
8
```

Modular arithmetic (Section 5) is commonly used in cryptography. The mod operator is the percent sign (be sure to set the scale back to 0 first):

```
scale=0
13 % 10
3
7 * 6 % 10
2
```

The real use of bc in this book comes when performing operators on large numbers. As bc is an arbitrary precision calculator, it will perform any calculation without approximating (although beware, some calculations will take a long time).

```
2^10
1024
2^100
1267650600228229401496703205376
2^1000
10715086071862673209484250490600018105614048117055336074437503883703\
51051124936122493198378815695858127594672917553146825187145285692314\
04359845775746985748039345677748242309854210746050623711418779541821\
53046474983581941267398767559165543946077062914571196477686542167660\
429831652624386837205668069376
2^10000000
...
```

While there are faster algorithms than what bc uses, it can be used for modular exponentiation.

```
29401^19231
11791936741673782277951361412655628509750802626058442595065879112837\
30645660979602186783941907308557893020948598603221372351480244103370\
...
27540919410894776657722419140083914356072020143002078956241640716425\
8780942697921463043977245290785757602091887914O1
29401^19231 % 37669
35694
```

To exit bc, type quit:

```
quit
```

To perform (normal) logarithms, you need to start bc using the -l option to load the math library. Then the l() function can be used to calculate the natural logarithm, or find the logarithm in any base. Although be careful with the scale.

```
$ bc -l
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free
    Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
<kbd>l(2.718)</kbd>
.99989631572895196894
<kbd>l(100)/l(10)</kbd>
2.00000000000000000000
<kbd>l(32)/l(2)</kbd>
5.00000000000000000004
<kbd>l(1024)/l(2)</kbd>
10.00000000000000000010
```

Discrete logarithms are not directly supported in bc.

3.1.3 Random Numbers

Random numbers are important for creating shared secret keys (as well as other use in other cryptographic operations). There are different ways to generate a random value in Linux. Two approaches are demonstrated in the following; a third approach is to use OpenSSL (introduced in Section 3.2 and demonstrated in Section 3.2.4.

Generating Random Numbers with Bash

The Bash shell has a built-in random number generator, which is accessed from the shell variable $RANDOM. It uses a Linear Congruential Generator (LCG) to return a value between 0 and 32,767. This is not a cryptographically strong Pseudo Random Number Generator (PRNG) and should *not* be used to create keys.

```
$ echo $RANDOM
4086
$ echo $RANDOM
11809
$ echo $RANDOM
6018
```

To see the details of the LCG algorithm used, look in the Bash source code; after downloading and unpacking the source, look in the file `variables.c`, search for the function `brand`. You can also see that the seed is based on the current time and process ID.

Generating Random Numbers with /dev/urandom

The Linux kernel has a pseudo-device `/dev/urandom` which is considered cryptographically strong PRNG for most applications. The device produces a continuous stream of random bytes, so while it is possible to view the stream in real-time using `cat`, it is common to pipe the output to select a specific number of bytes in an easy to read format. We can use `xxd` to do this.

First grab 8 Bytes, output in binary:

```
$ cat /dev/urandom | xxd -l 8 -b
0000000: 10000111 11110111 01001101 10011100 01111110 10110110 ..M.~.
0000006: 01010110 11010001
```

If we want 16 Bytes of hex output:

```
$ cat /dev/urandom | xxd -l 16 -g 16
00000000: 75619f0688497b213c5db43d49210c4d ua...I{!<].=I!.M
```

A little bit of text processing will return just the random value (omitting the other output produced by `xxd`). Let's use `cut` to grab the 2nd field, considering the output as space separated/delimited:

```
$ cat /dev/urandom | xxd -l 16 -g 16 | cut -d " " -f 2
313be197c436bebf074a2da3599a0ce0
```

Read the man pages for an explanation of the Linux kernel random number source device `/dev/urandom` and the related `/dev/random`. The section 7 man page gives an overview, while the section 4 man page gives more technical details on the two devices.

```
$ man -S7 random
$ man -S4 urandom
```

3.1.4 Hash Functions

Linux usually includes several commands for applying common hash functions on data. These are the “sum” commands, i.e. used for calculating checksums. The following commands demonstrate applications of the hash functions MD5, SHA1 and SHA2 (256).

```
$ cat demo.txt
This is a super secret message. $
$ md5sum demo.txt
7899f47eb650b40ae9156f6664304281 demo.txt
$ sha1sum demo.txt
87992f407ab94d05f64131db482067f9ffe42044 demo.txt
$ sha256sum demo.txt
12e38182116f070ef1a4d8961692787aa57add87d5496c4daf402279bc71c0b6 demo.txt
```

You can write the hash value to a file, and then use that file to perform a check:

```
$ sha256sum demo.txt > demo.sha256
$ cat demo.sha256
12e38182116f070ef1a4d8961692787aa57add87d5496c4daf402279bc71c0b6 demo.txt
$ sha256sum -c demo.sha256
demo.txt: OK
```

A change to the file should result in failure of the check (if the hash is not recomputed):

```
$ cat demo.txt
This is a super secret message. $
$ echo -n "This is a super secret message! " > demo.txt
$ sha256sum -c demo.sha256
demo.txt: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
```

You can also use OpenSSL (Section 3.2) to apply hash functions.

3.1.5 Bash Scripts

A useful advantage of the (Linux) command line is that it is easy to write a series of commands into a file, and then running them all by executing the file. This file is called a script. Once you have a basic grasp of the line command line (see Chapter 4 of my other book Network and Security in Linux), you can then start writing scripts to automate tasks. The basics of scripting, with many examples, is covered in Chapter 6 of my other book Network and Security in Linux.

3.2 OpenSSL

3.2.1 Overview of OpenSSL

https://www.openssl.org/ is a program and library that supports many different cryptographic operations, including:

- Symmetric key encryption
- Public/private key pair generation
- Public key encryption
- Hash functions
- Certificate creation
- Digital signatures
- Random number generation

While the primary purpose of OpenSSL is as a library, i.e. you write software that calls OpenSSL to perform cryptographic operations for your software, it also is a standalone program with a command-line interface. While we only use the standalone program, once you are familiar with it, you should be able to use the library.

OpenSSL supports different operations or commands, with the name of the command following `openssl`. For example, to perform symmetric key encryption the command is `enc` and on the command line you run:

```
$ openssl enc
```

Each of the operations supported by OpenSSL have a variety of options, such as input/output files, algorithms, algorithm parameters and formats. To start learning the details of OpenSSL, read the man page, i.e. `man openssl`. You'll soon learn that each of the operations (or commands) have their own man pages. For example, the operation of symmetric key encryption is `enc`, which is described in `man enc`.

There are other websites that give an overview of OpenSSL operations, as well as programming with the API. Check them out for more details.

3.2.2 Common Operations

OpenSSL takes an operation or command as first input, and that command may have it's own set of parameters. Parameters are usually specified starting with a dash (-). To see all the commands available, use the `help` command:

```
$ openssl help
Standard commands
asn1parse       ca              ciphers         cms
...

Message Digest commands (see the ‘dgst’␣command␣for␣more␣details)
blake2b512␣␣␣␣␣␣␣␣blake2s256␣␣␣␣␣␣␣␣gost␣␣␣␣␣␣␣␣␣␣␣␣␣␣md4
```

```
...

Cipher␣commands␣(see␣the␣‘enc’ command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
...
```

In reverse order:

Cipher commands perform symmetric key encryption

Message Digest commands apply hash functions

Standard commands provide a variety of operations related to public key generation and key management

For cipher and message digest commands, you can read the common format in the man pages `man enc` and `man dgst`, respectively. Most of the standard commands have their own man page, e.g. `man rsa`, `man x509`. Note that there are sometimes multiple commands that can be used to perform the same cryptographic operation. For example, you can generate Rivest Shamir Adleman cipher (RSA) key pairs using either `genrsa` or `genpkey` commands. This is mainly for compatibility reasons, that is, over time new commands have been added and the old command maintained.

Some common commands you will see in this book include:

- `enc` for symmetric key encryption
- `genpkey` for generating public/private key pairs
- `pkey` for processing and output keys from key pairs
- `dgst` for hashes, signatures and verification

3.2.3 Listing Ciphers and Algorithms

To see the (symmetric key) ciphers and (hash functions or) digests supported by your version of OpenSSL, you can use the `list` command. OpenSSL distinguishes between the *algorithms* and the *commands* used to call those algorithms. The following shows the version of OpenSSL, then lists algorithms and commands for ciphers and digests.

```
$ openssl version
OpenSSL 1.1.1 11 Sep 2018
$ openssl list -cipher-algorithms
AES-128-CBC
AES-128-CBC-HMAC-SHA1
...
SM4-ECB
SM4-OFB
$ openssl list -cipher-commands
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-ofb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-ofb
```

```
aria-256-cbc    aria-256-cfb    aria-256-cfb1   aria-256-cfb8
aria-256-ctr    aria-256-ecb    aria-256-ofb    base64
bf              bf-cbc          bf-cfb          bf-ecb
bf-ofb          camellia-128-cbc camellia-128-ecb camellia-192-cbc
camellia-192-ecb camellia-256-cbc camellia-256-ecb cast
cast-cbc        cast5-cbc       cast5-cfb       cast5-ecb
cast5-ofb       des             des-cbc         des-cfb
des-ecb         des-ede         des-ede-cbc     des-ede-cfb
des-ede-ofb     des-ede3        des-ede3-cbc    des-ede3-cfb
des-ede3-ofb    des-ofb         des3            desx
rc2             rc2-40-cbc      rc2-64-cbc      rc2-cbc
rc2-cfb         rc2-ecb         rc2-ofb         rc4
rc4-40          seed            seed-cbc        seed-cfb
seed-ecb        seed-ofb        sm4-cbc         sm4-cfb
sm4-ctr         sm4-ecb         sm4-ofb

openssl list -digest-algorithms
RSA-MD4 => MD4
RSA-MD5 => MD5
...
ssl3-sha1 => SHA1
whirlpool
$ openssl list -digest-commands
blake2b512      blake2s256      gost            md4
md5             rmd160          sha1            sha224
sha256          sha3-224        sha3-256        sha3-384
sha3-512        sha384          sha512          sha512-224
sha512-256      shake128        shake256        sm3
```

3.2.4 Random Number Generation with OpenSSL

Section 3.1.3 shows different ways to generate random numbers in Linux. OpenSSL has its own PRNG which is also considered cryptographically strong. This is accessed using the `rand` command and specifying the number of bytes to generate. To get hex output, use the `-hex` option:

```
$ openssl rand -hex 8
89978d4960720a750f35d569bcf28494
```

You can also output to a file and view the file with `xxd`:

```
$ openssl rand -out rand1.bin 8
$ ls -l rand1.bin
-rw-rw-r-- 1 sgordon sgordon 8 Jul 31 15:14 rand1.bin
$ xxd rand1.bin
0000000: 7d12 162f 1a18 c331                }../...1
$ xxd -b -g 8 -c 8 rand1.bin | cut -d " " -f 2
0111110100010010000101100010111100011010000110001100001100110001
```

On Linux, the OpenSSL `rand` command normally uses output from `/dev/urandom` to seed (initialise) it's PRNG. Read the man page for more information.

3.3 Python

Python (www.python.org) is a programming language that is seeing increasing use in networking applications that require cryptography. We use it for examples in this book as it is relatively quick to pick up and start building prototype applications with custom or existing cryptographic mechanisms.

Like most programming languages, including Java, PHP and C++, libraries are available that already implement common cryptographic mechanisms; you can focus your efforts on developing applications, not implementing encryption ciphers and hash algorithms. However there is currently no single standard cryptography library for Python; several are available. In this book we use the `cryptography` package, as introduced in Section 3.3.1. Another common library, not used in this book, is based on `PyNaCL` (https://pynacl.readthedocs.io/), which is based on libsodium and NaCL.

To use classical ciphers, the `PyCipher` package is used, which is introduced in Section 3.3.2.

3.3.1 Cryptography Package

For installation and quick usage guide, see https://cryptography.io/.

3.3.2 PyCipher Package

To learn some of the concepts and approaches used by current encryption algorithms (ciphers), it can be useful to first study how some of the original, simpler ciphers work (e.g. Caesar cipher, Playfair, Vigenere). With these simpler ciphers, often referred to as *classical ciphers*, it is quite easy to understand the algorithm and even perform encryption/decryption by hand. Although it is valuable to initially perform the encryption steps by hand, sometimes its useful to use software to speed things up. pycipher is a Python package that implements many classical ciphers. It has good documentation on how to use it, including installation instructions. Below I give two alternatives to install pycipher in a virtnet node. The first is the default and easiest that uses `git`. The second is an alternative if git is not available and you want a specific version of pycipher.

Install pycipher (Recommended Method)

In a terminal in Linux run:

```
$ sudo apt-get update
$ sudo apt-get install git python-pip
$ sudo pip install git+git://github.com/jameslyons/pycipher
```

Install pycipher (Alternative Method)

If the recommended method above does not work (e.g. you don't have or want to use `git` or `pip`), then you could try the following:

```
$ sudo apt-get install unzip python-setuptools
$ wget https://github.com/jameslyons/pycipher/archive/master.zip
$ unzip master.zip
$ cd pycipher-master/
$ sudo python setup.py install
$ python setup.py test
```

This installs and tests the latest version. Depending on the version, some tests my fail. In my case it ran 41 tests, but 2 tests failed (using the Porta algorithm). Do not use the algorithms that failed the tests.

Using pycipher

A quick example of encrypting and decrypting with pycipher is below. Other ciphers include: Beaufort, Foursquare, Enigma, Polybius, Bifid, ADFGVX, Coltrans, Playfair, and Vigenere. Details on the ciphers supported and how to use them are in the latest documentation.

```
$ python
Python 2.7.3 (default, Feb 27 2014, 20:00:17)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycipher
>>> pycipher.Caesar(3).encipher("hello")
'KHOOR'
>>> pycipher.Caesar(3).decipher("khoor")
'HELLO'
>>> quit()
```

Chapter 4

Statistics for Communications and Security

This chapter presents a selection of definitions and examples of mathematical properties that may be useful in learning computer communications and security.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

4.1 Binary Values

Applying several properties of exponentials and logarithms can make it easier when dealing with large binary values. Consider the following properties:

$$n^x \times n^y = n^{x+y}$$

$$\frac{n^x}{n^y} = n^{x-y}$$

$$\log_n(x \times y) = \log_n(x) + \log_n(y)$$

$$\log_n\left(\frac{x}{y}\right) = \log_n(x) - \log_n(y)$$

Example 4.1 (Properties of Exponentials)**.** Properties can be applied to simplify calculations:

$$\begin{aligned} 2^{12} &= 2^{2+10} \\ &= 2^2 \times 2^{10} \\ &= 4 \times 1024 \\ &= 4096 \end{aligned}$$

With this property of exponentials, if you can remember the values of 2^1 to 2^{10} then you can approximate most values of 2^b that you come across in communications and security. Table 4.1 gives the exact or approximate decimal value for b-bit numbers.

File: crypto/statistics.tex, r1791

Exponent, b	2^b	
(bits)	Exact Value	Approx. Value
0	1	-
1	2	-
2	4	-
3	8	-
4	16	-
5	32	-
6	64	-
7	128	-
8	256	-
9	512	-
10	1,024	$1{,}000 = 10^3$
11	-	2,000
12	-	4,000
13	-	8,000
14	-	16,000
	...	
19	-	512,000
20	-	$1{,}000{,}000 = 10^6$
21	-	2×10^6
22	-	4×10^6
23	-	8×10^6
	...	
29	-	512×10^6
30	-	10^9
31	-	2×10^9
32	-	4×10^9
33	-	8×10^9
	...	
39	-	512×10^9
40	-	10^{12}
50	-	10^{15}
60	-	10^{18}
70	-	10^{21}
$x \times 10$	-	10^{3x}

Table 4.1: Useful Exact and Approximate Values in Binary

Example 4.2 (Properties of Exponentials with Binary Values)**.** Properties and approximations can be used to perform large calculations:

$$\begin{aligned}\frac{2^{128}}{2^{100}} &= 2^{128-100} \\ &= 2^{28} \\ &= 2^{8} \times 2^{20} \\ &\approx 256 \times 10^{6} \\ &\approx 10^{8}\end{aligned}$$

Example 4.3 (Properties of Logarithms)**.** The number of bits needed to represent a decimal number can be found using logarithms:

$$\begin{aligned}\log_2(20,000) &= \log_2(20 \times 10^3) \\ &= \log_2(20) + \log_2(10^3) \\ &\approx 4 + 10 \\ &\approx 14\end{aligned}$$

4.2 Counting

Definition 4.1 (Number of Binary Values)**.** Given an n-bit number, there are 2^n possible values.

Example 4.4 (Number of Sequence Numbers)**.** Consider a sliding-window flow control protocol that uses an 16-bit sequence number. There are $2^{16} = 65,536$ possible values of the sequence number, ranging from 0 to 65,535 (after which it wraps back to 0).

Example 4.5 (Number of IP Addresses)**.** An IP address is a 32-bit value. There are 2^{32} or approximately 4×10^9 possible IP addresses.

Example 4.6 (Number of Keys)**.** If choosing a 128-bit encryption key randomly, then there are 2^{128} possible values of the key.

Video
Number of Binary Values (5 min; Jan 2015)
https://www.youtube.com/watch?v=AJU0BgwkXLU

Definition 4.2 (Fixed Length Sequences)**.** Given a set of n items, there are n^k possible k-item sequences, assuming repetition is allowed.

Example 4.7 (Sequences of PINs)**.** A user chooses a 4-digit PIN for a bank card. As there are 10 possible digits, there are 10^4 possible PINs to choose from.

Example 4.8 (Sequences of Keyboard Characters)**.** A standard keyboard includes 94 printable characters (a–z, A–Z, 0–9, and 32 punctuation characters). If a user must select a password of length 8, then there are 94^8 possible passwords that can be selected.

Video
Fixed Length Sequences (7 min; Jan 2015)
https://www.youtube.com/watch?v=9srF2V1f1gU

Definition 4.3 (Pigeonhole Principle)**.** If n objects are distributed over m places, and if $n > m$, then some places receive at least two objects.

Video
Pigeonhole Principle (2 min; Jan 2015)
https://www.youtube.com/watch?v=sz9yPCGW2D4

Example 4.9 (Pigeonhole Principle on Balls)**.** There are 20 balls to be placed in 5 boxes. At least one box will have at least two balls. If the balls are distributed in a uniform random manner among the boxes, then on average there will be 4 balls in each box.

Video
Pigeonhole Principle with Uniform Random Distribution (1 min; Jan 2015)
https://www.youtube.com/watch?v=PDCuL_SExu0

Example 4.10 (Pigeonhole Principle on Hash Functions)**.** A hash function takes a 100-bit input value and produces a 64-bit hash value. There are 2^{100} possible inputs distributed to 2^{64} possible hash values. Therefore at least some input values will map to the same hash value, that is, a collision occurs. If the hash function distributes the input values in a uniform random manner, then on average, there will be $\frac{2^{100}}{2^{64}} \approx 6.4 \times 10^{10}$ different input values mapping to the same hash value.

Video
Pigeonhole Principle and Hash Functions (5 min; Jan 2015)
https://www.youtube.com/watch?v=5xjMuZIMLLk

4.3 Permutations and Combinations

Definition 4.4 (Factorial)**.** There are $n!$ different ways of arranging n distinct objects into a sequence.

Example 4.11 (Factorial and Balls)**.** Consider four coloured balls: Red, Green, Blue and Yellow. There are $4! = 24$ arrangements (or permutations) of those balls:

```
RGBY, RGYB, RBGY, RBYG, RYGB, RYBG,
GRBY, GRYB, GBRY, GBYR, GYRB, GYBR,
BRGY, BRYG, BGRY, BGYR, BYRG, BYGR,
YRGB, YRBG, YGRB, YGBR, YBRG, YBGR
```

Video
Factorial and arranging balls (2 min; Jan 2015)
https://www.youtube.com/watch?v=Ay_E8bsOXJw

Example 4.12 (Factorial and English Letters)**.** The English alphabetic has 26 letters, a–z. There are $26! \approx 4 \times 10^{26}$ ways to arrange those 26 letters.

Video
Arranging English Letters (2 min; Jan 2015)
https://www.youtube.com/watch?v=ksilZXfwuQs

Example 4.13 (Factorial and Plaintext Messages)**.** An encryption algorithm takes a 64-bit plaintext message and a key as input and then maps that to a 64-bit ciphertext message as output. There are $2^{64} \approx 1.6 \times 10^{19}$ possible input plaintext messages. There are $2^{64}! \approx 10^{10^{88}}$ different reversible mappings from plaintext to ciphertext, i.e. $2^{64}!$ possible keys.

Video
Number of keys for ideal block cipher (6 min; Jan 2015)
https://www.youtube.com/watch?v=iQBLbz0w99s

Definition 4.5 (Combinations)**.** The number of combinations of items when selecting k at a time from a set of n items, assuming repetition is not allowed and order doesn't matter, is:

$$\frac{n!}{k!\,(n-k)!}$$

The following definition is just a specific instance of number of combinations (Definition 4.5) when $k = 2$. However the formula is simplified.

Definition 4.6 (Number of Pairs)**.** The number of pairs of items in a set of n items, assuming repetition is not allowed and order doesn't matter, is:

$$\frac{n\,(n-1)}{2}$$

Example 4.14 (Pairs of Coloured Balls)**.** There are four coloured balls: Red, Green, Blue and Yellow. The number of different coloured pairs of balls is $4 \times 3/2 = 6$. They are: `RG, RB, RY, GB, GY, BY`. Repetitions are not allowed (as they won't produce different coloured pairs), meaning `RR` is not a valid pair. Ordering doesn't matter, meaning `RG` is the same as `GR`.

Example 4.15 (Pairs of Network Devices)**.** A computer network has 10 devices. The number of links needed to create a full-mesh topology is $10 \times 9/2 = 45$.

Example 4.16 (Pairs of Key Sharers)**.** There are 50 users in a system, and each user shares a single secret key with every other user. The number of keys in the system is $50 \times 49/2 = 1,225$.

Video
Number of Pairs from n Items (5 min; Jan 2015)
https://www.youtube.com/watch?v=ZykkvK_Hu5g

4.4 Probability

In this chapter when referring to a "random" number it means taken from a uniform random distribution. That means there is equal probability of selecting each value from the set.

Definition 4.7 (Probability of Selecting a Value)**.** Probability of randomly selecting a specific value from a set of n values is $1/n$.

Example 4.17 (Probability of Selecting Coloured Ball)**.** There are five coloured balls in a box: red, green, blue, yellow and black. The probability of selecting the yellow ball is 1/5.

Example 4.18 (Probability of Selecting Backoff Value)**.** IEEE 802.11 (WiFi) involves a station selecting a random backoff from 0 to 15. The probability of selecting 5 is 1/16.

Video
Probability of Selecting a Particular Value from a Set (2 min; Jan 2015)
https://www.youtube.com/watch?v=hB5Hs4QPUUQ

Definition 4.8 (Total Expectation)**.** For a set of n events which are mutually exclusive and exhaustive, where for event i the expected value is E_i given probability P_i, then the total expected value is:

$$E = \sum_{i=1}^{n} E_i P_i$$

Video
Total Expectation Definition (1 min; Jan 2015)
https://www.youtube.com/watch?v=HiHIE9oFeiU

Example 4.19 (Total Expectation of Packet Delay)**.** Average packet delay for packets in a network is 100 ms along path 1 and 150 ms along path 2. Packets take path 1 30% of the time, and take path 2 70% of the time. The average packet delay across both paths is: $100 \times 0.3 + 150 \times 0.7 = 135$ ms.

Video
Total Expectation and Packet Delay (3 min; Jan 2015)
https://www.youtube.com/watch?v=-yxbhR-EeHQ

Example 4.20 (Total Expectation of Password Length)**.** In a network with 1,000 users, 150 users choose a 6-character password, 500 users choose a 7-character password, 250 users choose 9-character password and 100 users choose a 10-character password. The average password length is 7.65 characters.

Video
Total Expectation and Password Selection (3 min; Jan 2015)
https://www.youtube.com/watch?v=zTX7ENu-F20

Definition 4.9 (Number of Attempts)**.** If randomly selecting values from a set of n values, then the number of attempts needed to select a particular value is:
best case: 1
worst case: n
average case: $n/2$

Video
Number of Attempts Needed to Randomly Select a Value (1 min; Jan 2015)
https://www.youtube.com/watch?v=brDlrkuiH50

Example 4.21 (Number of Attempts in Choosing Number)**.** One person has chosen a random number between 1 and 10. Another person attempts to guess the random number. The best case is that they guess the chosen number on the first attempt. The worst case is that they try all other numbers before finally getting the correct number, that is 10 attempts. If the process is repeated 1000 times (that is, one person chooses a random number, the other guesses, then the person chooses another random number, and the other guesses again, and so on), then on average 10% of time it will take 1 attempt (best case), 10% of the time it will take 2 attempts, 10% of the time it will take 3 attempts, . . . , and 10% of the time it will take 10 attempts (worst case). The average number of attempts is therefore 5.

Video
Attempts to select a value between 1 and 10 (5 min; Jan 2015)
https://www.youtube.com/watch?v=nQUda8Uq-Ho

Example 4.22 (Number of Attempts in Choosing Key)**.** A user has chosen a random 128-bit encryption key. There are 2^{128} possible keys. It takes an attacker on average $2^{128}/2 = 2^{127}$ attempts to find the key. If instead a 129-bit encryption key was used, then the attacker would take on average $2^{129}/2 = 2^{128}$ attempts. (Increasing the key length by 1 bit doubles the number of attempts required by the attacker to guess the key).

Video
Attempts to guess a secret key (3 min; Jan 2015)
https://www.youtube.com/watch?v=8IttaYPN4MA

4.5 Collisions

Definition 4.10 (Birthday Paradox)**.** Given n random numbers selected from the range 1 to d, the probability that at least two numbers are the same is:

$$p(n;d) \approx 1 - \left(\frac{d-1}{d}\right)^{n(n-1)/2}$$

Example 4.23 (Two People Have Same Birthday)**.** Given a group of 10 people, the probability of at least two people have the same birth date (not year) is:

$$p(10;365) \approx 1 - \left(\frac{364}{365}\right)^{10(9)/2} = 11.6\%$$

Defintion 4.10 can be re-arranged to find the number of values needed to obtain a specified probability that at least two numbers are the same:

$$n(p;d) \approx \sqrt{2d \ln\left(\frac{1}{1-p}\right)}$$

Example 4.24 (Group Size for Birthday Matching)**.** How many people in a group are needed such that the probability of at least two of them having the same birth date is 50%?

$$n(0.5;365) \approx \sqrt{2 \times 365 \times \ln\left(\frac{1}{1-0.5}\right)} = 22.49$$

So 23 people in a group means there is 50% chance that at least two have the same birth date.

Example 4.25 (Group Size for Hash Collision)**.** Given a hash function that outputs a 64-bit hash value, how many attempts are need to give a 50% chance of a collision?

$$\begin{aligned} n(0.5;2^{64}) &\approx \sqrt{2 \times 2^{64} \times \ln\left(\frac{1}{1-0.5}\right)} \\ &\approx \sqrt{2^{64}} \\ &= 2^{32} \end{aligned}$$

Following Example 4.25, the number of attempts to produce a collision when using an n-bit hash function is approximately $2^{n/2}$.

Chapter 5

Number Theory

This chapter introduces basic concepts of number theory. These concepts are useful when studying several aspects of cryptography, especially public key cryptosystems.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

5.1 Divisibility and Primes

Definition 5.1 (Divides)**.** b *divides* a if $a = mb$ for some m, where a, b and m are integers. We can also say b is a *divisor* of a, or $b|a$.

Example 5.1 (Divides)**.** 3 divides 12, since $12 = 4 \times 3$. Also, 3 is a divisor of 12, or $3|12$.

Definition 5.2 (Greatest Common Divisor)**.** $\gcd(a, b)$ returns the greatest common divisor of integers a and b. There are efficient algorithms for finding the gcd, i.e. Euclidean algorithm.

Example 5.2 (Greatest Common Divisor)**.** $\gcd(12, 20) = 4$, since the divisors of 12 are (1, 2, 3, *4*, 6, 12) and the divisors of 20 are (1, 2, *4*, 5, 10, 20).

Definition 5.3 (Relatively Prime)**.** Two integers, a and b, are relatively prime if $\gcd(a, b) = 1$.

Example 5.3 (Relatively Prime)**.** $\gcd(7, 12) = 1$, since the divisors of 7 are (1, 7) and the divisors of 12 are (1, 2, 3, 4, 6, 12). Therefore 7 and 12 are relatively prime to each other.

Exercise 5.1 (Relatively Prime)**.** How many positive integers less than 10 are relatively prime with 10?

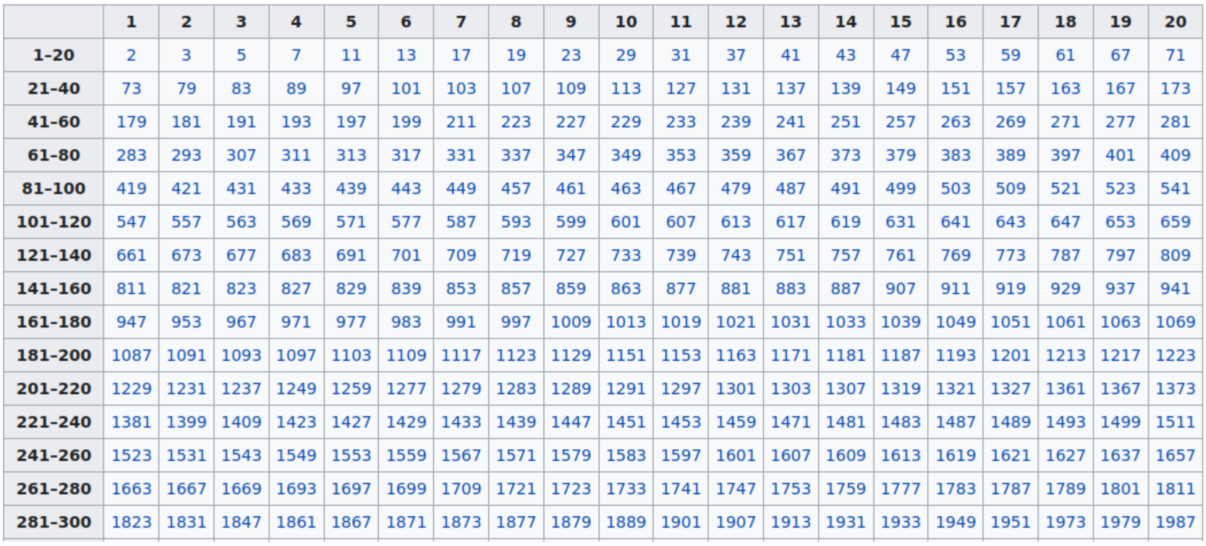
Solution 5.1 (Relatively Prime)**.** There are 9 positive integers less than 10, i.e. $1, 2, 3, \dots, 9$. For an integer a to be relatively prime to 10, then $\gcd(a, 10) = 1$. The divisors of 10 are 1, 2, 5 and 10. As the even integers have a divisor of 2, then they cannot be relatively prime with 10. That leaves 1, 3, 5, 7 and 9. $\gcd(5, 10) = 5$ and therefore 5 is not relatively prime with 10. The integers 1, 3, 7 and 9 cannot be divided by 3, 5 or 10, and therefore all have a greatest common divisor with 10 of 1. Hence 1, 3, 7 and 9 are less than 10 and relatively prime with 10. The answer is 4.

File: crypto/number.tex, r1963

Video
Divisibility, Greatest Common Divisor and Relatively Prime (10 min; Apr 2021)
https://www.youtube.com/watch?v=c5t9WuP8C1w

Definition 5.4 (Prime Number)**.** An integer $p > 1$ is a *prime number* if and only if its only divisors are $+1$, -1, $+p$ and $-p$.

Example 5.4 (Prime Number)**.** The divisors of 13 are (1, 13), that is, 1 and itself. Therefore 13 is a prime number. The divisors of 15 are (1, 3, 5, 15). Since the divisors include numbers other than 1 and itself, 15 is not prime.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1-20	2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71
21-40	73	79	83	89	97	101	103	107	109	113	127	131	137	139	149	151	157	163	167	173
41-60	179	181	191	193	197	199	211	223	227	229	233	239	241	251	257	263	269	271	277	281
61-80	283	293	307	311	313	317	331	337	347	349	353	359	367	373	379	383	389	397	401	409
81-100	419	421	431	433	439	443	449	457	461	463	467	479	487	491	499	503	509	521	523	541
101-120	547	557	563	569	571	577	587	593	599	601	607	613	617	619	631	641	643	647	653	659
121-140	661	673	677	683	691	701	709	719	727	733	739	743	751	757	761	769	773	787	797	809
141-160	811	821	823	827	829	839	853	857	859	863	877	881	883	887	907	911	919	929	937	941
161-180	947	953	967	971	977	983	991	997	1009	1013	1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
181-200	1087	1091	1093	1097	1103	1109	1117	1123	1129	1151	1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
201-220	1229	1231	1237	1249	1259	1277	1279	1283	1289	1291	1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
221-240	1381	1399	1409	1423	1427	1429	1433	1439	1447	1451	1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
241-260	1523	1531	1543	1549	1553	1559	1567	1571	1579	1583	1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
261-280	1663	1667	1669	1693	1697	1699	1709	1721	1723	1733	1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
281-300	1823	1831	1847	1861	1867	1871	1873	1877	1879	1889	1901	1907	1913	1931	1933	1949	1951	1973	1979	1987

Credit: Wikipedia, https://en.wikipedia.org/wiki/List_of_prime_numbers, CC BY-SA 3.0

Figure 5.1: First 300 Prime Numbers

Definition 5.5 (Prime Factors)**.** Any integer $a > 1$ can be factored as:

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

where $p_1 < p_2 < \ldots < p_t$ are prime numbers and where each a_i is a positive integer

Example 5.5 (Prime Factors)**.** The following are examples of integers expressed as prime factors:

$$\begin{aligned}
13 &= 13^1 \\
15 &= 3^1 \times 5^1 \\
24 &= 2^3 \times 3^1 \\
50 &= 2^1 \times 5^2 \\
560 &= 2^4 \times 5^1 \times 7^1 \\
2800 &= 2^4 \times 5^2 \times 7^1
\end{aligned}$$

Exercise 5.2 (Integers as Prime Factors)**.** Find the prime factors of 12870, 12936 and 30607.

Solution 5.2 (Integers as Prime Factors)**.** A naive approach (which works for these small examples) is to check if the number is divisible by primes, in increasing order. For example, is 12870 divisible by 2? Yes, then 2 is a prime factor. Is the result, 6435 divisible by 2? No, then is 6435 divisible by 3? Yes, and so on.

A "cheat" is to use software to find the factors. In Linux command line there is a command called `factor`.

The answers are:

$$\begin{gathered}12870 = 2^1 \times 3^2 \times 5^1 \times 11^1 \times 13^1 \\ 12936 = 2^3 \times 3^1 \times 7^2 \times 11^1 \\ 30607 = 127^1 \times 241^1\end{gathered}$$

Definition 5.6 (Prime Factorization Problem)**.** There are no known efficient, non-quantum algorithms that can find the prime factors of a sufficiently large number.

Example 5.6 (Prime Factorization Problem)**.** RSA Challenge involved researchers attempting to factor large numbers. Largest number measured in number of bits or decimal digits. Some records held over time are:

1991: 330 bits or 100 digits

2005: 640 bits or 193 digits

2009: 768 bits or 232 digits

Equivalent of 2000 years on single core 2.2 GHz computer to factor 768 bit

Current algorithms such as RSA rely on numbers of 1024, 2048 and even 4096 bits in length

Video
Prime Numbers and Prime Factorization (11 min; Apr 2021)
https://www.youtube.com/watch?v=i_LXZjK7Z98

Definition 5.7 (Euler's Totient Function)**.** Euler's totient function, $\phi(n)$, is the number of positive integers less than n and relatively prime to n. Also written as $\varphi(n)$ or $\text{Tot}(n)$.

Definition 5.8 (Properties of Euler's Totient)**.** Several useful properties of Euler's totient are:

$$\phi(1) = 1$$

$$\text{For prime } p, \phi(p) = p - 1$$

$$\text{For primes } p \text{ and } q, \phi(px \times q) = \phi(p) \times \phi(q) = (p - 1) \times (q - 1)$$

Example 5.7 (Euler's Totient Function)**.** The integers relatively prime to 10, and less than 10, are: 1, 3, 7, 9. There are 4 such numbers. Therefore $\phi(10) = 4$.

The integers relatively prime to 11, and less than 11, are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. There are 10 such numbers. Therefore $\phi(11) = 10$. The property could also be used since 11 is prime.

Since 7 is prime, $\phi(7) = 6$.

Since $77 = 7 \times 11$, then $\phi(77) = \phi(7 \times 11) = 6 \times 10 = 60$.

Video
Euler's Totient Function (8 min; Apr 2021)
https://www.youtube.com/watch?v=lnmbs-rPT-I

5.2 Modular Arithmetic

Definition 5.9 (Modular arithmetic simple)**.** Modular arithmetic is similar to normal arithmetic (addition, subtraction, multiplication, division) but the answers "wrap around".

Definition 5.10 (mod operator)**.** If a is an integer and n is a positive integer, then $a \bmod n$ is defined as the remainder when a is divided by n. n is called the *modulus*.

Example 5.8 (mod operator)**.** The following are several examples of mod:

$$3 \bmod 7 = 3, \text{ since } 0 \times 7 + 3 = 3$$

$$9 \bmod 7 = 2, \text{ since } 1 \times 7 + 2 = 9$$

$$10 \bmod 7 = 3, \text{ since } 1 \times 7 + 3 = 10$$

$$(-3) \bmod 7 = 4, \text{ since } (-1) \times 7 + 4 = -3$$

Definition 5.11 (Congruent modulo)**.** Two integers a and b are *congruent modulo n* if $(a \bmod n) = (b \bmod n)$. The congruence relation is written as:

$a \equiv b \pmod{n}$

When the modulus is known from the context, it may be written simply as a $\equiv b$.

Example 5.9 (Congruent modulo)**.** The following are examples of congruence:

$$3 \equiv 10 \pmod{7}$$

$$14 \equiv 4 \pmod{10}$$

$$3 \equiv 11 \pmod{8}$$

Definition 5.12 (Modular arithmetic)**.** Modular arithmetic with modulus n performs arithmetic operations within the confines of set $Z_n = \{0, 1, 2, \ldots, (n-1)\}$.

Example 5.10 (mod in Z_7)**.** Consider the set:

$$Z_7 = \{0, 1, 2, 3, 4, 5, 6\}$$

All modular arithmetic operations in mod 7 return answers in Z_7.

- If a is an integer and n is a positive integer, we define $a \bmod n$ to be the remainder when a is divided by n
- n is called the *modulus*
- Two integers a and b are *congruent modulo* n if $(a \bmod n) = (b \bmod n)$, which is written as
$$a \equiv b \pmod{n}$$
- $\pmod{n}$ operator maps all integers into the set of integers $Z_n = \{0, 1, \ldots, (n-1)\}$
- *Modular arithmetic* performs arithmetic operations within confines of set Z_n

Definition 5.13 (Modular Addition)**.** Addition in mod n is performed as normal addition, with the answer then mod by n.

Example 5.11 (Modular Addition)**.** The following are several examples of modular addition:

$$2+3 \pmod{7} = 5 \pmod{7} = 5 \bmod 7 = 5 \pmod{7}$$
$$2+6 \pmod{7} = 8 \pmod{7} = 8 \bmod 7 = 1 \pmod{7}$$
$$6+6 \pmod{7} = 12 \pmod{7} = 12 \bmod 7 = 5 \pmod{7}$$
$$3+4 \pmod{7} = 7 \pmod{7} = 7 \bmod 7 = 0 \pmod{7}$$

Definition 5.14 (Additive Inverse)**.** a is the *additive inverse* of b in mod n, if $a + b \equiv 0 \pmod{n}$.

For brevity, AI(a) may be used to indicate the additive inverse of a. One property is that all integers have an additive inverse.

Example 5.12 (Additive Inverse)**.** In mod 7:

$$AI(3) = 4, \text{ since } 3 + 4 \equiv 0 \pmod{7}$$

$$AI(6) = 1, \text{ since } 6 + 1 \equiv 0 \pmod{7}$$

In mod 12:

$$AI(3) = 9, \text{ since } 3 + 9 \equiv 0 \pmod{12}$$

Definition 5.15 (Modular Subtraction)**.** Subtraction in mod n is performed by addition of the additive inverse of the subtracted operand. This is effectively the same as normal subtraction, with the answer then mod by n.

Example 5.13 (Modular Subtraction)**.** For brevity, the modulus is sometimes omitted and $=$ is used in replace of $\equiv$. In mod 7:

$$6 - 3 = 6 + \text{AI}(3) = 6 + 4 = 10 = 3 \pmod 7$$

$$6 - 1 = 6 + \text{AI}(1) = 6 + 6 = 12 = 5 \pmod 7$$

$$1 - 3 = 1 + \text{AI}(3) = 1 + 4 = 5 \pmod 7$$

While the first two examples obviously give answers as we expect from normal subtraction, the third does as well. $1 - 3 = -2$, and in mod 7, $-2 \equiv 5$ since $-1 \times 7 + 5 = (-2)$. Recall $Z_7 = \{0, 1, 2, 3, 4, 5, 6\}$.

Video
Modular Addition, Additive Inverse and Modular Subtraction (13 min; Apr 2021)
https://www.youtube.com/watch?v=9uQe-7Fux9w

Definition 5.16 (Modular Multiplication)**.** Modular multiplication is performed as normal multiplication, with the answer then mod by n.

Example 5.14 (Modular Multiplication)**.** In mod 7:

$$2 \times 3 = 6 \pmod 7$$

$$2 \times 6 = 12 = 5 \pmod 7$$

$$3 \times 4 = 12 = 5 \pmod 7$$

Definition 5.17 (Multiplicative Inverse)**.** a is a multiplicative inverse of b in mod n if $a \times b \equiv 1 \pmod n$. For brevity, MI($a$) may be used to indicate the multiplicative inverse of a. a has a multiplicative inverse in (mod n) if a is relatively prime to n.

Example 5.15 (Multiplicative Inverse in mod 7)**.** 2 and 7 are relatively prime, therefore 2 has a multiplicative inverse in mod 7.

$$2 \times 4 \pmod 7 = 1, \text{ therefore } MI(2) = 4 \text{ and } MI(4) = 2$$

3 and 7 are relatively prime, therefore 3 has a multiplicative inverse in mod 7.

$$3 \times 5 \pmod 7 = 1, \text{ therefore } MI(3) = 5 \text{ and } MI(5) = 3$$

$\phi(7) = 6$, meaning 1, 2, 3, 4, 5 and 6 are relatively prime with 7, and therefore all of those numbers have a MI in mod 7.

Example 5.16 (Multiplicative Inverse in mod 8)**.** 3 and 8 are relatively prime, therefore 3 has a multiplicative inverse in mod 8.

$$3 \times 3 \pmod 8 = 1, \text{ therefore } MI(3) = 3$$

4 and 8 are NOT relatively prime, therefore 4 does not have a multiplicative inverse in mod 8. $\phi(8) = 4$, and therefore only 4 numbers (1, 3, 5, 7) have a MI in mod 8.

Definition 5.18 (Modular Division)**.** Division in mod n is performed as modular multiplication of the multiplicative inverse of 2nd operand. Modular division is only possible when the 2nd operand has a multiplicative inverse, otherwise the operation is undefined.

Example 5.17 (Modular Division)**.** In mod 7:

$$5 \div 2 = 5 \times MI(2) = 5 \times 4 = 20 \equiv 6$$

In mod 8:

$$7 \div 3 = 7 \times MI(3) = 7 \times 3 = 21 \equiv 5$$

$7 \div 4$ is undefined, since 4 does not have a multiplicative inverse in mod 8.

Definition 5.19 (Properties of Modular Arithmetic)**.**

$$(a \bmod n) \bmod n = a \bmod n$$

$$[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$$
$$[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$$
$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

Commutative, associative and distributive laws similar to normal arithmetic also hold.

Video
Modular Multiplication, Multiplicative Inverse (6 min; Apr 2021)
https://www.youtube.com/watch?v=hh0Nb_Gp-w0

5.3 Fermat's and Euler's Theorems

Definition 5.20 (Fermat's Theorem 1)**.** If p is prime and a is a positive integer not divisible by p, then:

$$a^{p-1} \equiv 1 \pmod p$$

Definition 5.21 (Fermat's Theorem 2)**.** If p is prime and a is a positive integer, then:

$$a^{p} \equiv a \pmod p$$

There are two forms of Fermat's theorem—use whichever form is most convenient.

Example 5.18 (Fermat's theorem)**.** What is 27^{42} mod 43? Since 43 is prime and $42 = 43 - 1$, this matches Fermat's Theorem form 1. Therefore the answer is 1.

Example 5.19 (Fermat's theorem)**.** What is 640^{163} mod 163? Since 163 is prime, this matches Fermat's Theorem form 2. Therefore the answer is 640, or simplified to 640 mod $163 = 151$.

Definition 5.22 (Euler's Theorem 1)**.** For every a and n that are relatively prime:

$$a^{\phi(n)} \equiv 1 \pmod n$$

Definition 5.23 (Euler's Theorem 2)**.** For positive integers a and n:

$$a^{\phi(n)+1} \equiv a \pmod{n}$$

Note that there are two forms of Euler's theorem—use the most relevant form.

Example 5.20 (Euler's theorem)**.** Show that $37^{40} \bmod 41 = 1$. Since $n = 41$, which is prime, then $\phi(41) = 40$. As 37 is also prime, 37 and 41 are relatively prime. Therefore Euler's Theorem form 1 holds.

Example 5.21 (Euler's theorem)**.** What is $13794^{4621} \bmod 4757$? Factoring 4757 into primes gives 67×71. Therefore $\phi(4757) = \phi(67)x \times \phi(71) = 66 \times 70 = 4620$. Therefore, this follows Euler's Theorem form 2, giving an answer of 13794.

Video
Fermat's and Euler's Theorems (16 min; Apr 2021)
https://www.youtube.com/watch?v=k0NBKQ8W90U

5.4 Discrete Logarithms

Definition 5.24 (Modular Exponentiation)**.** As exponentiation is just repeated multiplication, modular exponentiation is performed as normal exponentiation with the answer mod by n.

Example 5.22 (Modular Exponentiation)**.**

$$2^3 \bmod 7 = 8 \bmod 7 = 1$$

$$3^4 \bmod 7 = 81 \bmod 7 = 4$$

$$3^6 \bmod 8 = 729 \bmod 8 = 1$$

Video
Modular Exponentiation (1 min; Apr 2021)
https://www.youtube.com/watch?v=Nj6GxMeYRO4

Definition 5.25 (Normal Logarithm)**.** If $b = a^i$, then:

$$i = \log_a(b)$$

read as "the log in base a of b is index (or exponent) i".

The above definition is for normal arithmetic, not for modular arithmetic. Logarithm in normal arithmetic is the inverse operation of exponentiation. In modular arithmetic, modular logarithm is more commonly called *discrete logarithm*. Note we replace n with p—the reason will become apparent shortly.

Definition 5.26 (Discrete Logarithm)**.** If $b = a^i \pmod{p}$, then:

$$i = \text{dlog}_{a,p}(b)$$

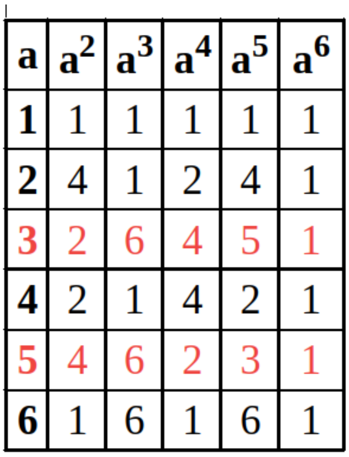
A unique exponent i can be found if a is a *primitive root* of the prime p.

Video
Normal and Discrete Logarithms (3 min; Apr 2021)
https://www.youtube.com/watch?v=T17nhLEWwoA

Definition 5.27 (Primitive Root)**.** If a is a primitive root of prime p then $a_1, a_2, a_3, \ldots a_{p-1}$ are distinct in mod p.

The integers with a primitive root are: 2, 4, p^α, $2p^\alpha$ where p is any odd prime and α is a positive integer.

Example 5.23 (Primitive Root)**.** Consider the prime $p = 7$:
$a = 1 : 1^2 \bmod 7 = 1, 1^3 \bmod 7 = 1, ...$(not distinct)
$a = 2 : 2^2 \bmod 7 = 4, 2^3 \bmod 7 = 1, 2^4 \bmod 7 = 2, 2^5 \bmod 7 = 4, ...$(not distinct)
$a = 3 : 3^2 \bmod 7 = 2, 3^3 \bmod 7 = 6, 3^4 \bmod 7 = 4, 3^5 \bmod 7 = 5, 3^6 \bmod 7 = 1$(distinct)
Therefore 3 is a primitive root of 7 (but 1 and 2 are not).



a	**a²**	**a³**	**a⁴**	**a⁵**	**a⁶**
1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

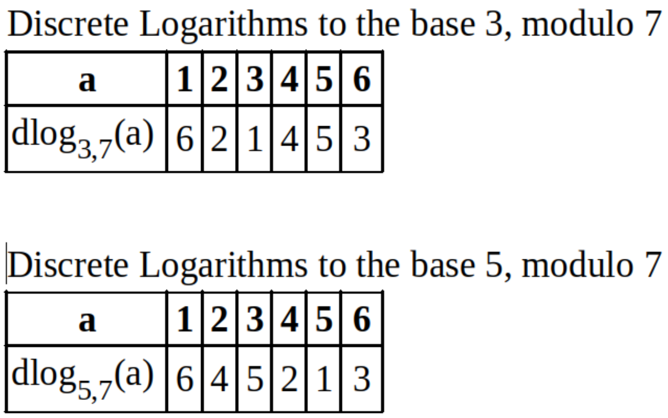
Figure 5.2: Powers of Integers, modulo 7

From the above table we see 3 and 5 are primitive roots of 7.

Discrete logarithms to the base 3, modulo 7 are distinct since 3 is a primitive root of 7. Discrete logarithms to the base 5, modulo 7 are distinct since 5 is a primitive root of 7.

We see that 3, 5, 6, 7, 10, 11, 12 and 14 are primitive roots of 17.

The discrete logarithm in modulo 17 can be calculated for the 8 primitive roots.



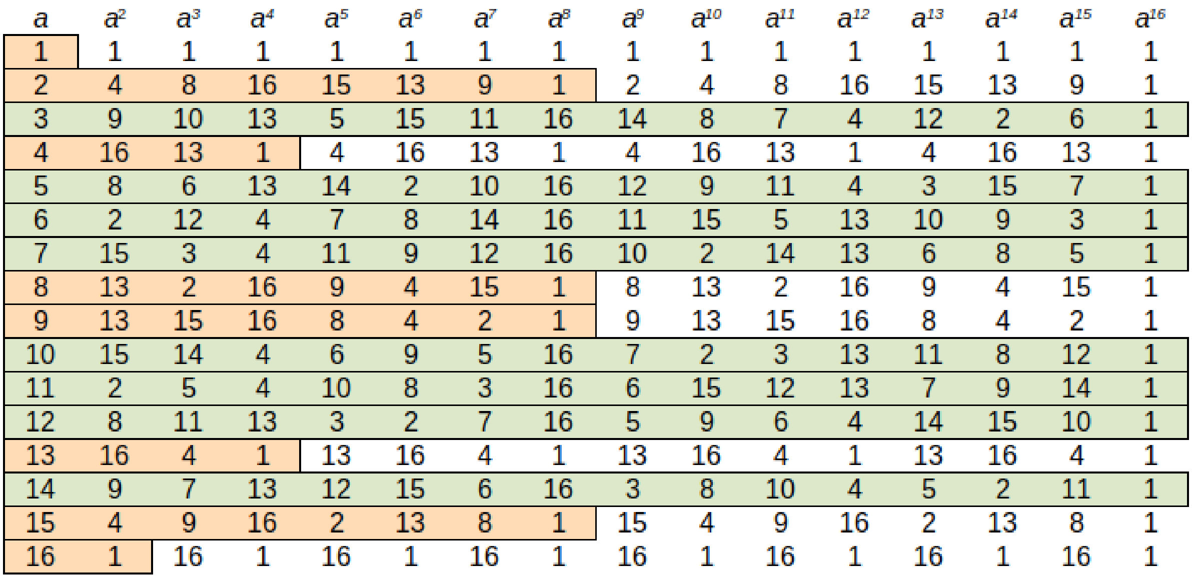
Discrete Logarithms to the base 3, modulo 7

a	**1**	**2**	**3**	**4**	**5**	**6**
$\mathrm{dlog}_{3,7}(a)$	6	2	1	4	5	3

Discrete Logarithms to the base 5, modulo 7

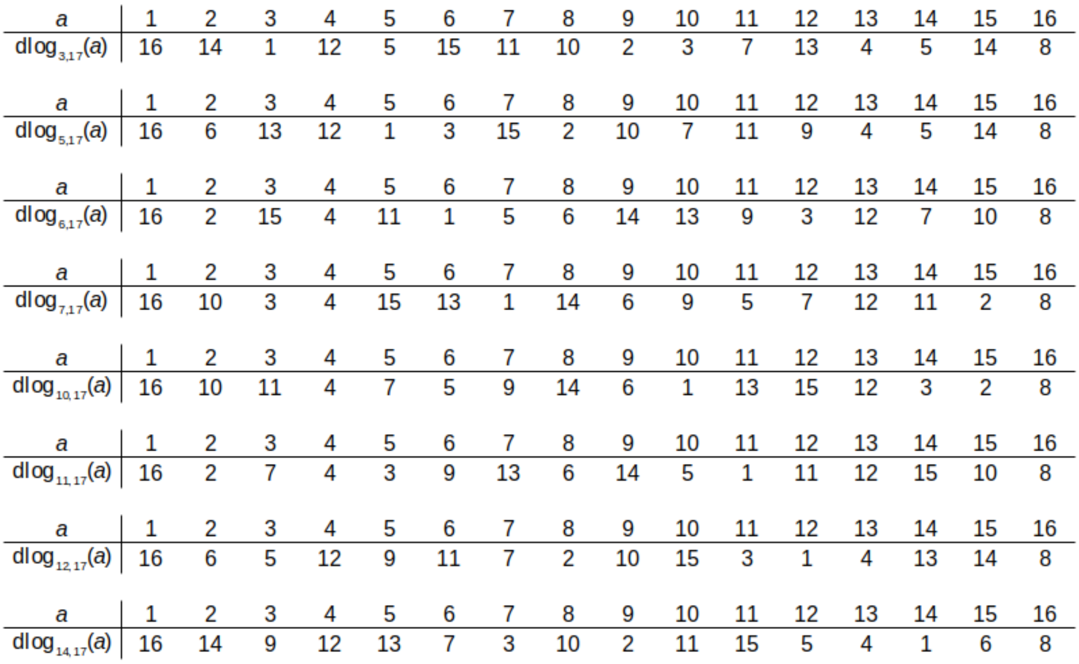
a	**1**	**2**	**3**	**4**	**5**	**6**
$\mathrm{dlog}_{5,7}(a)$	6	4	5	2	1	3

Figure 5.3: Discrete Logs, modulo 7



a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	15	13	9	1	2	4	8	16	15	13	9	1
3	9	10	13	5	15	11	16	14	8	7	4	12	2	6	1
4	16	13	1	4	16	13	1	4	16	13	1	4	16	13	1
5	8	6	13	14	2	10	16	12	9	11	4	3	15	7	1
6	2	12	4	7	8	14	16	11	15	5	13	10	9	3	1
7	15	3	4	11	9	12	16	10	2	14	13	6	8	5	1
8	13	2	16	9	4	15	1	8	13	2	16	9	4	15	1
9	13	15	16	8	4	2	1	9	13	15	16	8	4	2	1
10	15	14	4	6	9	5	16	7	2	3	13	11	8	12	1
11	2	5	4	10	8	3	16	6	15	12	13	7	9	14	1
12	8	11	13	3	2	7	16	5	9	6	4	14	15	10	1
13	16	4	1	13	16	4	1	13	16	4	1	13	16	4	1
14	9	7	13	12	15	6	16	3	8	10	4	5	2	11	1
15	4	9	16	2	13	8	1	15	4	9	16	2	13	8	1
16	1	16	1	16	1	16	1	16	1	16	1	16	1	16	1

Figure 5.4: Powers of Integers, modulo 17



a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{3,17}(a)$	16	14	1	12	5	15	11	10	2	3	7	13	4	5	14	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{5,17}(a)$	16	6	13	12	1	3	15	2	10	7	11	9	4	5	14	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{6,17}(a)$	16	2	15	4	11	1	5	6	14	13	9	3	12	7	10	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{7,17}(a)$	16	10	3	4	15	13	1	14	6	9	5	7	12	11	2	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{10,17}(a)$	16	10	11	4	7	5	9	14	6	1	13	15	12	3	2	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{11,17}(a)$	16	2	7	4	3	9	13	6	14	5	1	11	12	15	10	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{12,17}(a)$	16	6	5	12	9	11	7	2	10	15	3	1	4	13	14	8

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{dlog}_{14,17}(a)$	16	14	9	12	13	7	3	10	2	11	15	5	4	1	6	8

Figure 5.5: Discrete Logarithms, modulo 17

5.5 Computationally Hard Problems

There are several problems in number theory that are considered computationally hard. That means, when sufficiently large numbers are used, solving the problems are practically impossible. These computationally hard problems are used to provide security in cryptographic mechanisms, especially in public key cryptography. Three important problems considered impossible to solve with conventional computers follow.

Definition 5.28 (Hard Problem: Integer Factorisation)**.** If p and q are unknown primes, given $n = pq$, find p and q.

Also known as prime factorisation. While someone that knows p and q can easily calculate n, if an attacker knows only n they cannot find p and q.

Definition 5.29 (Hard Problem: Euler's Totient)**.** Given composite n, find $\phi(n)$.

While it is easy to calculate Euler's totient of a prime, or of the multiplication of two primes if those primes are known, an attacker cannot calculate Euler's totient of sufficiently large non-prime number. Solving Euler's totient of n, where $n = pq$, is considered to be harder than integer factorisation.

Definition 5.30 (Hard Problem: Discrete Logarithms)**.** Given b, a, and p, find i such that $i = \mathrm{dlog}_{a,p}(b)$.

While modular exponentiation is relatively easy, such as calculating $b = a^i \bmod p$, the inverse operation of discrete logarithms is computationally hard. The complexity is considered comparable to that of integer factorisation.
When studying RSA and Diffie-Hellman, you will see how these hard problems in number theory are used to secure ciphers.

Video
Computationally Hard Problems for Cryptography (4 min; Apr 2021)
https://www.youtube.com/watch?v=fJXdXNxeNVs

Part III

Symmetric Key Encryption

Chapter 6

Classical Ciphers

This chapter introduces several historical or classical ciphers. While these ciphers are no longer used, they are simple enough to perform operations by hand while demonstrating important concepts used in the design of most symmetrical ciphers used today. The actual history of the ciphers is not presented here; you can find that in most cryptography textbooks or via searches online.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

6.1 Caesar Cipher

6.1.1 Caesar Cipher Definitions and Examples

Algorithm 6.1 (Caesar Cipher)**.** To encrypt with a key `k`, shift each letter of the plaintext `k` positions to the right in the alphabet, wrapping back to the start of the alphabet if necessary. To decrypt, shift each letter of the ciphertext `k` positions to the left (wrapping if necessary).

In the examples we will assume the Caesar cipher (and most other classical ciphers) operate on case-insenstive English plaintext. That is, the character set is a through to z. However it can also be applied to any language or character set, so long as the character set is agreed upon by the users.

Exercise 6.1 (Caesar Cipher Encryption)**.** Using the Caesar cipher, encrypt plaintext `hello` with key `3`.

Solution 6.1 (Caesar Cipher Encryption)**.** To encrypt the plaintext `hello` with the key `3`, each letter in the plaintext is encrypted by shifting 3 positions to the right in the alphabet. The letter 3 positions to the right of `h` is `K`, as illustrated below:

```
a b c d e f g h i j K l m n o p q r s t u v w x y z
```

The letter 3 positions to the right of `e` is `H`:

```
a b c d e f g H i j k l m n o p q r s t u v w x y z
```

File: crypto/classical.tex, r1964

The letter 3 positions to the right of `l` is `O` (notin that there are two `l`'s in the plaintext, so there will be two `O`'s in the ciphertext):

`a b c d e f g h i j k l m n O p q r s t u v w x y z`

The letter 3 positions to the right of `o` is `R`:

`a b c d e f g h i j k l m n o p q R s t u v w x y z`

The final ciphertext is therefore `KHOOR`.

Video
Caesar Cipher Encryption Example (2 min; Feb 2020)
https://www.youtube.com/watch?v=HILcygVamnU

Question 6.1 (How many keys are possible in the Caesar cipher?)**.** If the Caesar cipher is operating on the characters a–z, then how many possible keys are there? Is a key of `0` possible? Is it a good choice? What about a key of `26`?

Video
Number of Keys in Caesar Cipher (3 min; Feb 2020)
https://www.youtube.com/watch?v=Uk1k_GA_2Y0

Exercise 6.2 (Caesar Cipher Decryption)**.** You have received the ciphertext `TBBQOLR`. You know the Caesar cipher was used with key `n`. Find the plaintext.

Solution 6.2 (Caesar Cipher Decryption)**.** To decrypt the ciphertext `TBBQOLR` with the key `n`, each letter in the ciphertext is decrypted by shifting n=13 positions to the left in the alphabet. The letter 13 positions to the left of `T` is `g`, as illustrated below:

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

The letter 13 positions to the left of `B` is `o`:

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

Therefore, the first three letters of the plaintext so far are `goo`. You can continue as above to find the final plaintext is `goodbye`.

Video
Caesar Cipher Decryption Example (3 min; Feb 2020)
https://www.youtube.com/watch?v=N6YwWnkXh8M

We will now look at the Caesar cipher from a mathematical perspective. By treating each letter in the alphabet as a number, we can write equations that define the encrypt and decrypt operations on each letter.

Algorithm 6.2 (Caesar Cipher, formal)**.**

$$C = E(K, P) = (P + K) \bmod 26 \tag{6.1}$$

$$P = D(K, C) = (C - K) \bmod 26 \tag{6.2}$$

In the equations, P is the numerical value of a plaintext letter. Letters are numbered in alphabetical order starting at 0. That is, a=0, b=1, ..., z=25. Similarly, K and C are the numerical values of the key and ciphertext letter, respectively. Shifting to the right in encryption is addition, while shifting to the left in decryption is subtraction. To cater for the wrap around (e.g. when the letter `z` is reacher), the last step is to mod by the total number of characters in the alphabet.

Exercise 6.3 (Caesar Cipher, formal)**.** Consider the following mapping.

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

Use the the formal (mathematical) algorithm for Caesar cipher to decrypt `SDV` with key `p`.

Solution 6.3 (Caesar Cipher Encryption)**.** Key `p` means $K = 15$. The first ciphertext letter is `S`, so $C_1 = 18$. Using the decrypt equation:

$$\begin{aligned} P_1 &= (C_1 - K) \bmod 26 \\ &= (18 - 15) \bmod 26 \\ &= 3 \bmod 26 \\ &= 3 \end{aligned}$$

Therefore the first plaintext letter is `d`.

The same decrypt equation and key are used for the second ciphertext letter of `D`, i.e. $C_2 = 3$.

$$\begin{aligned} P_2 &= (C_2 - K) \bmod 26 \\ &= (3 - 15) \bmod 26 \\ &= (-12) \bmod 26 \\ &= 14 \end{aligned}$$

Therefore the second plaintext letter is `o`.

The same decrypt equation and key are used for the third ciphertext letter of `V`, i.e. $C_3 = 21$.

$$\begin{aligned} P_3 &= (C_3 - K) \bmod 26 \\ &= (21 - 15) \bmod 26 \\ &= (6) \bmod 26 \\ &= 6 \end{aligned}$$

Therefore the third plaintext letter is g, and the entire plaintext is dog.

Video
Caesar Cipher Decryption using Mathematical Approach (4 min; Feb 2020)
https://www.youtube.com/watch?v=yvLYP7zxnkA

Listing 6.1: Caesar Encrypt and Decrypt

```
>>> pycipher.Caesar(3).encipher("hello")
'KHOOR'
>>> pycipher.Caesar(3).decipher("khoor")
'HELLO'
```

Note that the pycipher package needs to be installed and imported first (see Section 3.3.2).

6.1.2 Brute Force Attack on Caesar Cipher

Definition 6.1 (Brute Force Attack)**.** Try all combinations (of keys) until the correct plaintext/key is found.

Exercise 6.4 (Caesar Brute Force)**.** The ciphertext FRUURJVBCANNC was obtained using the Caesar cipher. Find the plaintext using a brute force attack.

Solution 6.4. As a naive approach, try all possible keys, and then check the plaintext values obtained. If one is recognisable, then most likely have found the correct plaintext. Without any knowledge of which key was used, one approach is to try the keys in order. For example, try key 1, and then key 2, then key 3. (In theory you could try key 0, but we know in the Caesar cipher that it does nothing).

Listing 6.2: Caesar Brute Force

```
for k in range(0,26):
  pycipher.Caesar(k).decipher("FRUURJVBCANNC")
```

The range function in Python produces values inclusive of the lower limit and exclusive of the upper limit. That is, from 0 to 25.

Listing 6.3: Caesar Brute Force Results

```
0: FRUURJVBCANNC 13: SEHHEWIOPNAAP
1: EQTTQIUABZMMB 14: RDGGDVHNOMZZO
2: DPSSPHTZAYLLA 15: QCFFCUGMNLYYN
3: CORROGSYZXKKZ 16: PBEEBTFLMKXXM
4: BNQQNFRXYWJJY 17: OADDASEKLJWWL
5: AMPPMEQWXVIIX 18: NZCCZRDJKIVVK
6: ZLOOLDPVWUHHW 19: MYBBYQCIJHUUJ
7: YKNNKCOUVTGGV 20: LXAAXPBHIGTTI
8: XJMMJBNTUSFFU 21: KWZZWOAGHFSSH
9: WILLIAMSTREET 22: JVYYVNZFGERRG
10: VHKKHZLRSQDDS 23: IUXXUMYEFDQQF
11: UGJJGYKQRPCCR 24: HTWWTLXDECPPE
12: TFIIFXJPQOBBQ 25: GSVVSKWCDBOOD
```

The results of the brute force are formatted to show the key (it is slightly different from the Python code output).

Video
Brute Force Attack on Caesar Cipher with Python (5 min; Feb 2020)
https://www.youtube.com/watch?v=GpfoaxcxHWs

Question 6.2 (How many attempts for Caesar brute force?)**.** What is the worst, best and average case of number of attempts to brute force ciphertext obtained using the Caesar cipher?

There are 26 letters in the English alphabet. The key can therefore be one of 26 values, 0 through to 25. The key of 26 is equivalent to a key of 0, since it will encrypt to the same ciphertext. The same applies for all values greater than 25. While a key of 0 is not very smart, let's assume it is a valid key.

The best case for the attacker is that the first key they try is the correct key (i.e. 1 attempt). The worst case is the attacker must try all the wrong keys until they finally try the correct key (i.e. 26 attempts). Assuming the encrypter chose the key randomly, there is equal probability that the attacker will find the correct key in 1 attempt (1/26), as in 2 attempts (1/26), as in 3 attempts (1/26), and as in 26 attempts (1/26). The average number of attempts can be calculated as $(26+1)/2 = 13.5$.

Assumption 6.1 (Recognisable Plaintext upon Decryption)**.** The decrypter will be able to recognise that the plaintext is correct (and therefore the key is correct). Decrypting ciphertext using the incorrect key will *not* produce the original plaintext. The decrypter will be able to recognise that the key is wrong, i.e. the decryption will produce unrecognisable output.

Question 6.3 (Is plaintext always recognisable?)**.** Caesar cipher is using recognisably correct plaintext, i.e. English words. But is the correct plaintext always recognisable? What if the plaintext was a different language? Or compressed? Or it was an image or video? Or binary file, e.g. .exe? Or a set of characters chosen randomly, e.g. a key or password?

The correct plaintext is recognisable if it contains some structure. That is, it does not appear random. It is common in practice to add structure to the plaintext, making it relatively easy to recognise the correct plaintext. For example, network packets have headers/trailers or error detecting codes. Later we will see cryptographic mechanisms that can be used to ensure that the correct plaintext will be recognised. For now, let's assume it can be.

There are two ways to improve the Caesar cipher:

1. Increase the key space so brute force is harder

2. Change the plaintext (e.g. compress it) so harder to recognise structure

6.2 Monoalphabetic Ciphers

6.2.1 Monoalphabetic Cipher Definitions and Examples

Definition 6.2 (Permutation)**.** A permutation of a finite set of elements is an ordered sequence of all the elements of S, with each element appearing exactly once. In general, there are $n!$ permutations of a set with n elements.

The concept of permutation is used throughput cryptography, and shortly we will see in a monoalphabetic (substitution) cipher.

Example 6.1 (Permutation)**.** Consider the set $S = \{a, b, c\}$. There are six permutations of S:

```
abc, acb, bac, bca, cab, cba
```

This set has 3 elements. There are $3! = 3 \times 2 \times 1 = 6$ permutations.

Definition 6.3 (Monoalphabetic (Substitution) Cipher)**.** Given the set of possible plaintext letters (e.g. English alphabetc, a–z), a single permutation is chosen and used to determine the corresponding ciphertext letter.

This is a monoalphabetic cipher because only a single cipher alphabet is used per message.

Example 6.2 (Monoalphabetic (Substitution) Cipher)**.** In advance, the sender and receiver agree upon a permutation to use, e.g.:

```
P: a b c d e f g h i j k l m n o p q r s t u v w x y z
C: H P W N S K L E V A Y C X O F G T B Q R U I D J Z M
```

To encrypt the plaintext `hello`, the agreed upon permutation (or mapping) is used to produce the ciphertext `ESCCF`.

Exercise 6.5 (Decrypt Monoalphabetic Cipher)**.** Decrypt the ciphertext `QSWBSR` using the permutation chosen in the previous example.

Solution 6.5 (Decrypt Monoalphabetic Cipher)**.** A simple lookup on the mapping defined in the example returns the plaintext `secret`.

Video
Mono-alphabetic Substitution Cipher Example (3 min; Feb 2020)
https://www.youtube.com/watch?v=RWoDvO2WQ0A

Question 6.4 (How many keys in English monoalphabetic cipher?)**.** How many possible keys are there for a monoalphabetic cipher that uses the English lowercase letters? What is the length of an actual key?

Consider the number of permutations possible. The example used a single permutation chosen by the two parties.

Video
Number of Keys in an English Monoalphabetic Substitution Cipher (3 min; Feb 2020)
https://www.youtube.com/watch?v=XXCHks0vMW0

6.2.2 Brute Force Attack on Monoalphabetic Cipher

Exercise 6.6 (Brute Force on Monoalphabetic Cipher)**.** You have intercepted a ciphertext message that was obtained with an English monoalphabetic cipher. You have a Python function called:
`mono_decrypt_and_check(ciphertext,key)`
that decrypts the ciphertext with a key, and returns the plaintext if it is correct, otherwise returns false. You have tested the Python function in a while loop and the computer can apply the function at a rate of 1,000,000,000 times per second. Find the average time to perform a brute force on the ciphertext.

Solution 6.6 (Brute Force on Monoalphabetic Cipher)**.** With a 26 letter alphabet, there are 26! permutations or keys. The average number of keys to try in a brute force attack is $(26! + 1)/2$, or approximately half of them, $26!/2$. The Python code can try 10^9 keys per second. Therefore the average brute force time, T, is:

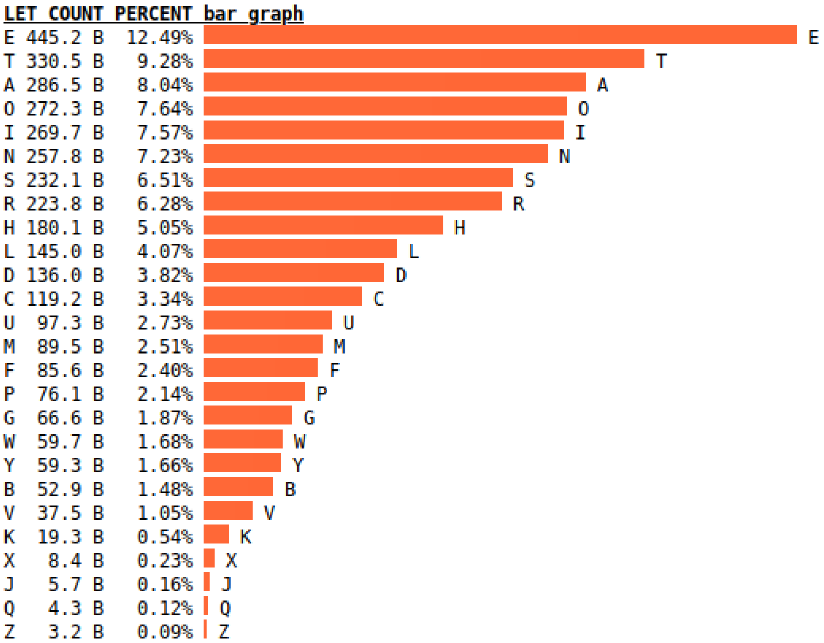
$$\begin{aligned} T &= \frac{(26! + 1)/2}{10^9} \\ &\approx \frac{2 \times 10^{26}}{10^9} \\ &\approx 2 \times 10^{17} \text{ seconds} \\ &\approx 64 \text{ million centuries} \end{aligned}$$

Video
Brute Force Attack Time on English Monoalphabetic Cipher (7 min; Feb 2020)
https://www.youtube.com/watch?v=c4gLyX9mwgM

6.2.3 Frequency Analysis Attack on Monoalphabetic Cipher

Brute force is the “dumb” approach to breaking a cipher. While it was sufficient in breaking the Caesar cipher, it is not feasible for a monoalphabetic substitution cipher. Can we take a “smart” approach that would take less effort than brute force? Often we can. Let’s consider frequency analysis as an alternative to a brute force attack.

Definition 6.4 (Frequency Analysis Attack)**.** Find (portions of the) key and/or plaintext by using insights gained from comparing the actual frequency of letters in the ciphertext with the expected frequency of letters in the plaintext. Can be expanded to analyse sets of letters, e.g. digrams, trigrams, n-grams, words.



LET	COUNT	PERCENT
E	445.2 B	12.49%
T	330.5 B	9.28%
A	286.5 B	8.04%
O	272.3 B	7.64%
I	269.7 B	7.57%
N	257.8 B	7.23%
S	232.1 B	6.51%
R	223.8 B	6.28%
H	180.1 B	5.05%
L	145.0 B	4.07%
D	136.0 B	3.82%
C	119.2 B	3.34%
U	97.3 B	2.73%
M	89.5 B	2.51%
F	85.6 B	2.40%
P	76.1 B	2.14%
G	66.6 B	1.87%
W	59.7 B	1.68%
Y	59.3 B	1.66%
B	52.9 B	1.48%
V	37.5 B	1.05%
K	19.3 B	0.54%
X	8.4 B	0.23%
J	5.7 B	0.16%
Q	4.3 B	0.12%
Z	3.2 B	0.09%

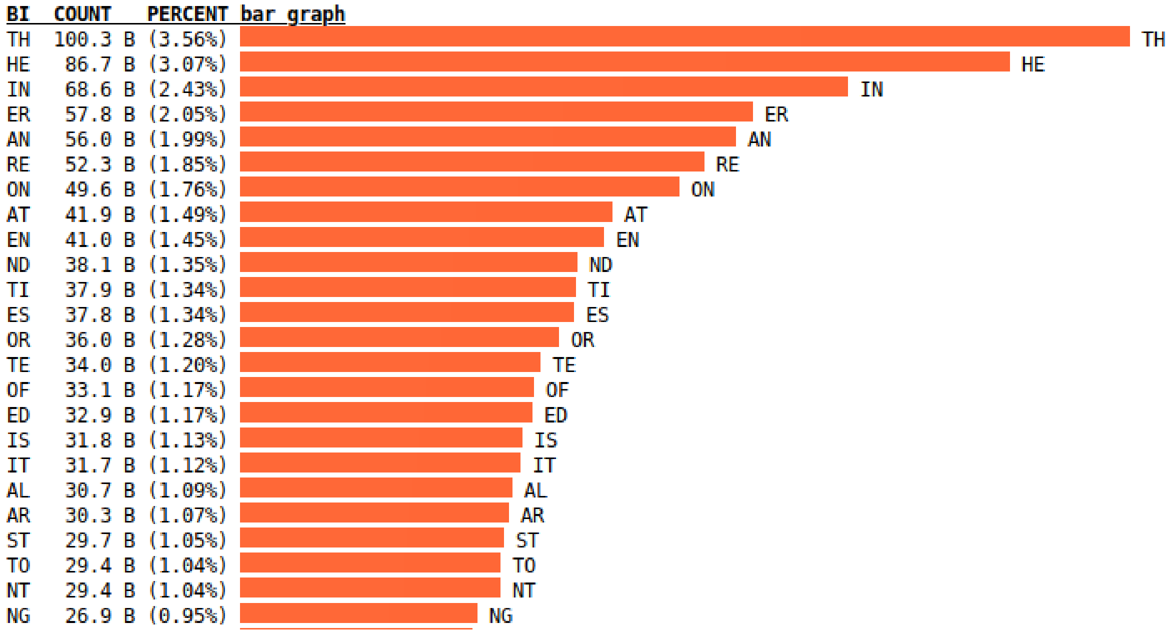
Credit: *Letter Counts* by Peter Norvig

Figure 6.1: Relative Frequency of Letters by Norvig

The letter frequencies of the figure above are based on Peter Norvig's analysis of Google Books N-Gram Dataset. Norvig is Director of Research at Google. His website has more details on the analysis.

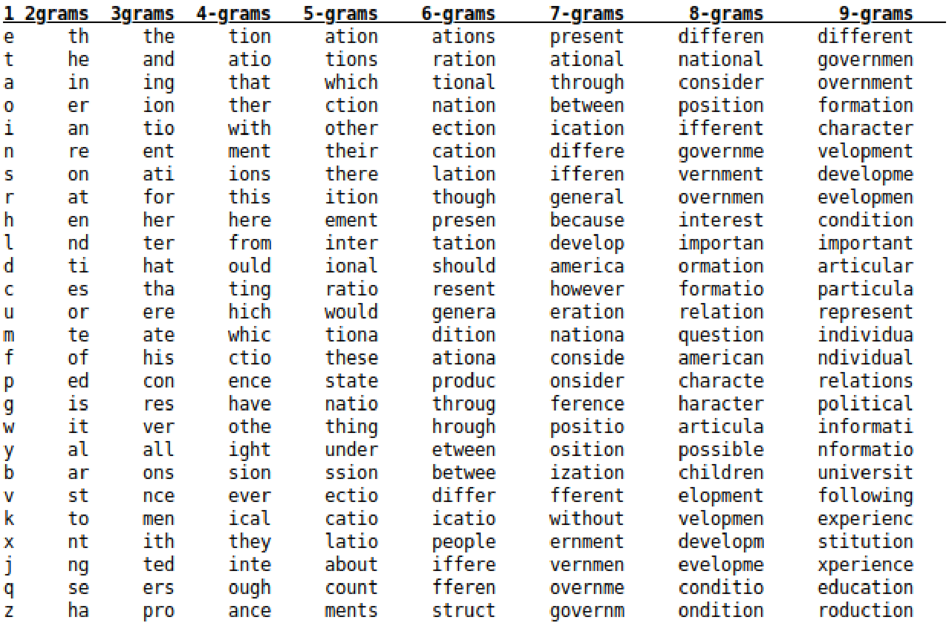
Exercise 6.7 (Break a Monoalphabetic Cipher)**.** Ciphertext:

```
ziolegxkltqodlzgofzkgrxetngxzgzithkofeohs
tlqfrzteifojxtlgyltexkofuegdhxztklqfregd
hxztkftzvgkalvoziygexlgfofztkftzltexkoznz
itegxkltoltyytezoctsnlhsozofzgzvghqkzlyo
klzofzkgrxeofuzitzitgkngyeknhzgukqhinofes
xrofuigvdqfnesqlloeqsqfrhghxsqkqsugkozid
lvgkaturtlklqrouozqsloufqzxktlqfrltegfrhk
gcorofurtzqoslgyktqsofztkftzltexkoznhkgz
gegslqsugkozidlqfrziktqzltuohltecokxltlyo
ktvqsslitfetngxvossstqkfwgzizitgktzoeqsq
lhtezlgyegdhxztkqfrftzvgkaltexkoznqlvtssq
ligvziqzzitgknolqhhsotrofzitofztkftzziol
afgvstrutvossitshngxofrtloufofuqfrrtctsgh
ofultexktqhhsoeqzogflqfrftzvgkahkgzgegsl
qlvtssqlwxosrofultexktftzvgkal
```



Credit: *Two-Letter Sequence (Bigram) Counts* by Peter Norvig

Figure 6.2: Relative Frequency of Digrams by Norvig



1	2grams	3grams	4-grams	5-grams	6-grams	7-grams	8-grams	9-grams
e	th	the	tion	ation	ations	present	differen	different
t	he	and	atio	tions	ration	ational	national	governmen
a	in	ing	that	which	tional	through	consider	overnment
o	er	ion	ther	ction	nation	between	position	formation
i	an	tio	with	other	ection	ication	ifferent	character
n	re	ent	ment	their	cation	differe	governme	velopment
s	on	ati	ions	there	lation	ifferen	vernment	developme
r	at	for	this	ition	though	general	overnmen	evelopmen
h	en	her	here	ement	presen	because	interest	condition
l	nd	ter	from	inter	tation	develop	importan	important
d	ti	hat	ould	ional	should	america	ormation	articular
c	es	tha	ting	ratio	resent	however	formatio	particula
u	or	ere	hich	would	genera	eration	relation	represent
m	te	ate	whic	tiona	dition	nationa	question	individua
f	of	his	ctio	these	ationa	conside	american	ndividual
p	ed	con	ence	state	produc	onsider	characte	relations
g	is	res	have	natio	throug	ference	haracter	political
w	it	ver	othe	thing	hrough	positio	articula	informati
y	al	all	ight	under	etween	osition	possible	nformatio
b	ar	ons	sion	ssion	betwee	ization	children	universit
v	st	nce	ever	ectio	differ	fferent	elopment	following
k	to	men	ical	catio	icatio	without	velopmen	experienc
x	nt	ith	they	latio	people	ernment	developm	stitution
j	ng	ted	inte	about	iffere	vernmen	evelopme	xperience
q	se	ers	ough	count	fferen	overnme	conditio	education
z	ha	pro	ance	ments	struct	governm	ondition	roduction

Credit: *N-Letter Sequences (N-grams)"* by Peter Norvig

Figure 6.3: Relative Frequency of N-Grams by Norvig

Solution 6.7 (Break a Monoalphabetic Cipher)**.** See the the steps under the section "Frequency Analysis of Monoalphabetic Cipher" on the following website:
sandilands.info/sgordon/classical-ciphers-frequency-analysis-examples

6.3 Playfair Cipher

Algorithm 6.3 (Playfair Matrix Construction)**.** Write the letters of keyword `k` row-by-row in a 5-by-5 matrix. Do not include duplicate letters. Fill the remainder of the matrix with the alphabet. Treat the letters i and j as the same (that is, they are combined in the same cell of the matrix).

Exercise 6.8 (Playfair Matrix Construction)**.** Construct the Playfair matrix using keyword `australia`.

Solution 6.8 (Playfair Matrix Construction)**.** We write the keyword in a 5-by-5 matrix, starting as:

```
a u s t r
l i
```

Note that we don't write the letter a multiple times in the matrix, and the letter i also represents the letter j.

Now we fill the remainder of the matrix with the English letters in alphabetical order. Again, no duplicate letters are included.

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

Video
Playfair Cipher Matrix Construction (3 min; Feb 2020)
https://www.youtube.com/watch?v=5QGiCkZidE4

Algorithm 6.4 (Playfair Encryption)**.** Split the plaintext into pairs of letters. If a pair has identical letters, then insert a special letter x in between. If the resulting set of letters is odd, then pad with a special letter x.

Locate the plaintext pair in the Playfair matrix. If the pair is on the same column, then shift each letter down one cell to obtain the resulting ciphertext pair. Wrap when necessary. If the plaintext pair is on the same row, then shift to the right one cell. Otherwise, the first ciphertext letter is that on the same row as the first plaintext letter and same column as the second plaintext letter, and the second ciphertext letter is that on the same row as the second plaintext letter and same column as the first plaintext letter.

Repeat for all plaintext pairs.

Playfair decryption uses the same matrix and reverses the rules. That is, move up (instead of down) if on the same column, move left (instead of right) if on the same row. Finally, the padded special letters need to be removed. This can be done based upon knowledge of the langauge. For example, if the intermediate plaintext from decryption is `helxlo`, then as that word doesn't exist, the x is removed to produce `hello`.

Exercise 6.9 (Playfair Encryption)**.** Find the ciphertext if the Playfair cipher is used with keyword `australia` and plaintext `hello`.

Solution 6.9 (Playfair Encryption)**.** The Playfair matrix from the previous exercise is:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

First split the plaintext into pairs: `he`, `ll` and `o`. As the second pair has identical letters, insert a special character x and move the second `l` into the third pair. The resulting pairs are:

```
he lx lo
```

Now for each pair, apply the rules to find the corresponding ciphertext pair.

For plaintext pair `he`:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

As the pair are on the same row, the ciphertext pair is taken as the letters to the right:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

The first two letters of the ciphertext are `KF`.

The second pair, `lx`, is on different rows and columns:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

The ciphertext pair is taken from the same row and column, but reversed in order:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

The second pair of ciphertext is `BV`.

Finally, the third pair:

```
a u s t r
l i b c d
e f g h k
m n o p q
v w x y z
```

As a result, the final ciphertext is KFBVBM.

Video
Encryption with the Playfair Cipher (7 min; Feb 2020)
https://www.youtube.com/watch?v=7kmTq35mLzA

Question 6.5 (Does Playfair cipher always map a letter to the same ciphertext letter?)**.** Using the Playfair cipher with keyword australia, encrypt the plaintext hellolove.

With the Playfair cipher, if a letter occurs multiple times in the plaintext, will that letter always encrypt to the same ciphertext letter?

If a pair of letters occurs multiple times, will that pair always encrypt to the same ciphertext pair?

Is the Playfair cipher subject to frequency analysis attacks?

Video
Playfair Cipher and Frequency Analysis (4 min; Feb 2020)
https://www.youtube.com/watch?v=gPmRhuXd5j0

6.4 Polyalphabetic Ciphers

Definition 6.5 (Polyalphabetic (Substitution) Cipher)**.** Use a different monoalphabetic substitution as proceeding through the plaintext. A key determines which monoalphabetic substitution is used for each transformation.

For example, when encrypting a set of plaintext letters with a polyalphabetic cipher, a monoalpabetic cipher with a particular key is used to encrypt the first letter, and then the same monoalphabetic cipher is used but with a different key to encrypt the second letter. They key used for the monoalphabetic cipher is determined by the key (or keyword) for the polyalphabetic cipher.

- Vigenère Cipher: uses Caesar cipher, but Caesar key changes each letter based on keyword
- Vernam Cipher: binary version of Vigenère, using XOR
- One Time Pad: same as Vigenère/Vernam, but random key as long as plaintext

Selected polyalphabetic ciphers are explained in depth in the following sections.

6.5 Vigenère Cipher

Algorithm 6.5 (Vigenère Cipher)**.** For each letter of plaintext, a Caesar cipher is used. The key for the Caesar cipher is taken from the Vigenère key(word), progressing for each letter and wrapping back to the first letter when necessary. Formally, encryption using a keyword of length m is:

$$c_i = (p_i + k_{i \bmod m}) \bmod 26$$

where p_i is letter i (starting at 0) of plaintext P, and so on.

Simply, Vigenère cipher is just the Caesar cipher, but changing the Caesar key for each letter encrypted/decrypted. The Caesar key is taken from the Vigenère key. The Vigenère key is not a single value/letter, but a set of values/letters, and hence referred to as a keyword. Encrypting the first letter of plaintext uses the first key from the keyword. Encrypting the second letter of plaintext uses the second key from the keyword. And so on. As the keyword (for convenience) is usually shorter than the plaintext, once the end of the keyword is reached, we return to the first letter, i.e. wrap around.

In the formal equation for encryption, i represents letter i (starting at 0) of the plaintext. For example, if the keyword is 6 letters, when encrypting letter 8 of the plaintext (that is the 9th), then k_2 is used, i.e. the 3rd letter from the keyword.

Example 6.3 (Vigenère Cipher Encryption)**.** Using the Vigenère cipher to encrypt the plaintext `carparkbehindsupermarket` with the keyword `sydney` produces the ciphertext `UYUCEPCZHUMLVQXCIPEYUXIR`. The keyword would be repeated when Caesar is applied:

```
P: carparkbehindsupermarket
K: sydneysydneysydneysydney
C: UYUCEPCZHUMLVQXCIPEYUXIR
```

Note that the first `a` in the plaintext transforms to `Y`, while the second `a` transforms to `E`. With polyalphabetic ciphers, the same plaintext letters do not necessarily always transform to the same ciphertext letters. Although they may: look at the third `a`.

Video
Encryption with Vigenere Cipher and Python (4 min; Feb 2020)
https://www.youtube.com/watch?v=r8xsofoAdNI

Exercise 6.10 (Vigenère Cipher Encryption)**.** Use Python (or other software tools) to encrypt the plaintext `centralqueensland` with the following keys with the Vigenère cipher, and investigate any possible patterns in the ciphertext: `cat`, `dog`, `a`, `giraffe`.

Solution 6.10 (Vigenère Cipher Encryption)**.** Using the `pycipher` library:

```
$ python3
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycipher
>>> pycipher.Vigenere("cat").encipher("centralqueensland")
'EEGVRTNQNGEGULTPD'
```

```
>>> pycipher.Vigenere("dog").encipher("centralqueensland")
'FSTWFGOEAHSTVZGQR'
>>> pycipher.Vigenere("a").encipher("centralqueensland")
'CENTRALQUEENSLAND'
>>> pycipher.Vigenere("giraffe").encipher("centralqueensland")
'IMETWFPWCVESXPGVU'
```

Video
Vigenere Python Examples (4 min; Feb 2020)
https://www.youtube.com/watch?v=VqjDjocUqKY

While the Vigenère cipher improves on monoalphabetic ciphers, it still has a weakness. The approach for breaking the cipher is:

- Determine the length of the keyword m
  - Repeated n-grams in the ciphertext may indicate repeated n-grams in the plaintext
  - Separation between repeated n-grams indicates possible keyword length m
  - If plaintext is long enough, multiple repetitions make it easier to find m
- Treat the ciphertext as that from m different monoalphabetic ciphers
  - E.g. Caesar cipher with m different keys
  - Break the monoalphabetic ciphers with frequency analysis
- With long plaintext, and repeating keyword, Vigenère can be broken

The following shows an example of breaking the Vigenère cipher, although it is not necessary to be able to do this yourself manually.

Example 6.4 (Breaking Vigenère Cipher)**.** Ciphertext `ZICVTWQNGRZGVTWAVZHCQYGLMGJ` has repetition of `VTW`. That suggests repetition in the plaintext at the same position, which would be true if the keyword repeated at the same position.

```
012345678901234567890123456
ZICVTWQNGRZGVTWAVZHCQYGLMGJ
```

That is, it is possible the key letter at position 3 is the repated at position 12. That in turn suggest a keyword length of 9 or 3.

```
ciphertext ZICVTWQNGRZGVTWAVZHCQYGLMGJ
length=3:  012012012012012012012012012
length=9:  012345678012345678012345678
```

An attacker would try both keyword lengths. With a keyword length of 9, the attacker then performs Caesar cipher frequency analysis on every 9th letter. Eventually they find plaintext is `wearediscoveredsaveyourself` and keyword is `deceptive`.

This attack may require some trial-and-error, and will be more likely to be successful when the plaintext is very long. See the Stallings textbook, from which the example is taken, for further explanation.

Video
Cryptanalysis of Vigenere Cipher (4 min; Feb 2020)
https://www.youtube.com/watch?v=u0FtomzkoTQ

6.6 Vernam Cipher

Before looking at an improvement of the Vigenère cipher, let's look at a cipher that is essentially the same but operates on binary data.

Algorithm 6.6 (Vernam Cipher)**.** Encryption is performed as:

$$c_i = p_i \oplus k_i$$

decryption is performed as:

$$p_i = c_i \oplus k_i$$

where p_i is the ith bit of plaintext, and so on. The key is repeated where necessary.

The Vernam cipher is essentially a binary form of the Vigenère cipher. The mathematical form of Vigenère encryption adds the plaintext and key and mods by 26 (where there are 26 possible charactersd). In binary, there are 2 possible characters, so the equivalnet is to add the plaintext and key and mod by 2. This identical to the XOR operation.

To demonstrate the Vernam cipher, we will use Python to perform the XOR ($\oplus$) operation.

Listing 6.4: XOR

```
>>> def xor(x, y):
...     return '{1:0{0}b}'.format(len(x), int(x, 2) ^ int(y, 2))
...
```

The Python code defines a function called **xor** that takes two strings representing bits, and returns a string represent the XOR of those bits. The actual XOR is performed on integers using the Python hat ôperator. The rest is formatting as strings.

Exercise 6.11 (Vernam Cipher Encryption)**.** Using the Vernam cipher, encrypt the plaintext 011101010101000011011001 with the key 01011.

Listing 6.5: Vernam Cipher Encryption

```
>>> xor('011101010101000011011001','010110101101011010110101')
'001011111000011001101100'
```

Video
Vernam Cipher using Bits and XOR (7 min; Feb 2020)
https://www.youtube.com/watch?v=oXsyZhZrRE4

6.7 One Time Pad

The weakness of the Vigenère and Vernam ciphers is a repeating keyword. The solution is to use a key as long as the plaintext and entirely random.

Algorithm 6.7 (One-Time Pad)**.** Use polyalphabetic cipher (such as Vigenère or Vernam) but where the key must be: random, the same length as the plaintext, and not used multiple times.

Essentially, the Vigenère or Vernam become a One-Time Pad (OTP) if the keys are chosen appropriately.

The result of using a long, random key is the OTP has the following properties:

- Encrypting plaintext with random key means output ciphertext will be random
  - E.g. XOR plaintext with a random key produces random sequence of bits in ciphertext
- Random ciphertext contains no information about the structure of plaintext
  - Attacker cannot analyse ciphertext to determine plaintext
- Brute force attack on key is ineffective
  - Multiple different keys will produce recognisable plaintext
  - Attacker has no way to determine which of the plaintexts are correct
- OTP is only known unbreakable (unconditionally secure) cipher

Example 6.5 (Attacking OTP)**.** Consider a variant of Vigenère cipher that has 27 characters (including a space). An attacker has obtained the ciphertext:

```
ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS
```

Attacker tries all possible keys. Two examples:

```
k1:  pxlmvmsydofuyrvzwc tnlebnecvgdupahfzzlmnyih
p1:  mr mustard with the candlestick in the hall
k2:  pftgpmiydgaxgoufhklllmhsqdqogtewbqfgyovuhwt
p2:  miss scarlet with the knife in the library
```

There are many other legible plaintexts obtained with other keys. No way for attacker to know the correct plaintext

The example shows that even a brute force attack on a OTP is unsuccessful. Even if the attacker could try all possible keys—the plaintext is 43 characters long and so there are $27^{43} \approx 10^{61}$ keys—they would find many possible plaintext values that make sense. The example shows two such plaintext values that the attacker obtained. Which one is the correct plaintext? They both make sense (in English). The attacker has no way of knowing. In general, there will be many plaintext values that make sense from a brute force attack, and the attacker has no way of knowing which is the correct (original) plaintext. Therefore a brute force attack on a OTP is ineffective.

Let's finish our coverage of classical substitition ciphers with a summary of the OTP:

- Only known unbreakable (unconditionally secure) cipher
  - Ciphertext has no statistical relationship with plaintext
  - Given two potential plaintext messages, attacker cannot identify the correct message
- But two significant practical limitations:
  1. Difficult to create large number of random keys
  2. Distributing unique long random keys is difficult
- Limited practical use

The practical limittions are significant. The requirement that the key must be as long as the plaintext, random and never repeated (if it is repeated then the same problems arise as in the original Vernam cipher) means large random values must be created. But creating a large amount of random data is actually difficult. Imagine you wanted to use a OTP for encrypting large data transfers (multiple gigabytes) across a network. Multiple gigabytes of random data must be generated for the key, which is time consuming (seconds to hours) for some computers. Also, the key must be exchanging, usually over a network, with the other party in advance. So to encrypt a 1GB file to need a 1GB random key. Both the key and file must be sent across the network, i.e. a total of 2GB. This is very inefficient use of the network: a maximum of 50% efficiency.

Later we will see real ciphers that work with a relatively small, fixed length key (e.g. 128 bits) and provide sufficient security.

Video
One-Time Pad as an Unbreakable Cipher (7 min; Feb 2020)
https://www.youtube.com/watch?v=GSsDofkajD4

6.8 Transposition Techniques

The previous set of classical ciphers use a substitution operation, replacing one character with another from the character set. A different approach is to simply re-arrange the set of characters within the plaintext. These type of ciphers are called transposition or permutation techniques.

- Substitution: replace one (or more) character in plaintext with another from the entire possible character set
- Transposition: re-arrange the characters in the plaintext
  - The set of characters in the ciphertext is the same as in the plaintext
  - Problem: the plaintext frequency statistics are also in the ciphertext
- On their own, transposition techniques are easy to break

- Combining transposition with substitution makes ciphers stronger, and building block of modern ciphers

Definition 6.6 (Rail Fence Cipher Encryption)**.** Select a depth as a key. Write the plaintext in diagonals in a zig-zag manner to the selected depth. Read row-by-row to obtain the ciphertext.

The decryption process can easily be derived from the encryption algorithm.

Exercise 6.12 (Rail Fence Encryption)**.** Consider the plaintext `securityandcryptography` with key `4`. Using the rail fence cipher, find the ciphertext.

Solution 6.11 (Rail Fence Encryption)**.** With a key of `4`, we write the plaintext in diagonals over 4 rows.

```
s     t     r     r
 e   i y   c y   g a
  c r   a d   p o   p y
   u     n     t     h
```

The ciphertext is obtained by reading row-by-row: `STRREIYEYGACRADPOPYUNTH`.

Video
Rail Fence Transposition Cipher Example (2 min; Feb 2020)
https://www.youtube.com/watch?v=35OM0S_MDcg

Definition 6.7 (Rows Columns Cipher Encryption)**.** Select a number of columns m and permutate the integers from 1 to m to be the key. Write the plaintext row-by-row over m columns. Read column-by-column, in order of the columns determined by the key, to obtain the ciphertext.

Be careful with the decryption process; it is often confusing. Of course it must be the process such that the original plaintext is produced.

Exercise 6.13 (Rows Columns Encryption)**.** Consider the plaintext `securityandcryptography` with key `315624`. Using the rows columns cipher, find the ciphertext.

Solution 6.12 (Rows Columns Encryption)**.** With a key of `315624`, we write the plaintext row-by-row across 6 columns:

```
3 1 5 6 2 4
s e c u r i
t y a n d c
r y p t o g
r a p h y x
```

A special letter, `x` in this case, is used to pad to fill the last row. This padding must be agreed upon in advance by the sender and receiver.

Now read column-by-column, starting with column indicated by the key as 1, i.e. `EYYA`. Then column 2: `RDOY`. The resulting ciphertext is `EYYARDOYSTRRICGXCAPPUNTH`.

Video
Encrypting with Rows/Columns Transposition Cipher (3 min; Feb 2020)
https://www.youtube.com/watch?v=nZdtNiDLKJ0

Example 6.6 (Rows Columns Multiple Encryption)**.** Assume the ciphertext from the previous example has been encrypted again with the same key. The resulting ciphertext is `YYCPRRCTEOIPDRAHYSGUATXH`. Now let's view how the cipher has "mixed up" the letters of the plaintext. If the plaintext letters are numbered by position from 01 to 24, their order (split across two rows) is:

```
01 02 03 04 05 06 07 08 09 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24
```

After first encryption the order becomes:

```
02 08 14 20 05 11 17 23 01 07 13 19
06 12 18 24 03 09 15 21 04 10 16 22
```

After the second encryption the order comes:

```
08 23 12 21 05 13 03 16 02 17 06 15
11 19 09 20 14 01 18 04 20 07 24 10
```

Are there any obviously obversvable patterns?

After the first encryption, the numbers reveal a pattern: increasing by 6 within groups of 4. This is because of the 6 columns and 4 rows. After the second encryption, it is not so obvious to identify patterns.

The point is that while a single application of the transposition cipher did not seem to offer much security (in terms of hiding patterns), adding the second application of the cipher offers an improvement. This principle of repeated applications of simple operations is used in modern ciphers.

In summary:

- Transposition ciphers on their own offer no practical security
- But combining transposition ciphers with substitution ciphers, and repeated applications, practical security can be achieved
- Modern symmetric ciphers use multiple applications (rounds) of substitition and transposition (permutation) operations

Video
Multiple Rounds of Rows/Columns Transposition Cipher (5 min; Feb 2020)
https://www.youtube.com/watch?v=oxQBHCWqe6A

Chapter 7

Encryption and Attacks

Chapter 6 introduced concepts of encryption using classical ciphers. This chapter formalises these concepts, in Section 7.1 defining the building blocks for encryption in modern ciphers, in particular in symmetric key cryptography. Section 7.2 looks at encryption from the attackers point of view. Understanding the approaches attackers can take is necessary to be able to build secure systems with will withstand attacks. Section 7.3 and Section 7.4 outline the general design approaches to the two types of symmetric key ciphers: block ciphers and stream ciphers.

Further details about encryption and attacks are covered in subsequent chapters, including details on Data Encryption Standard (DES) (Chapter 8) and Advanced Encryption Standard (AES) (Chapter 9). The alternative to symmetric key encryption, public key cryptography is introduced in Part IV.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

7.1 Encryption Building Blocks

Video
Encryption Building Blocks (13 min; Mar 2020)
https://www.youtube.com/watch?v=ZVF2kYPnm3g

Figure 7.1 shows the general model for encrypting for confidentiality that we have seen previously.

Symmetric sender/receiver use same key (single-key, secret-key, shared-key, conventional)

Public-key sender/receiver use different keys (asymmetric)

All ciphers until about the 1960's were symmetric key ciphers. The encrypter and decrypter used the same key, i.e. symmetry between the keys. The key must be shared between the two users and kept secret.

File: crypto/encryption.tex, r1965

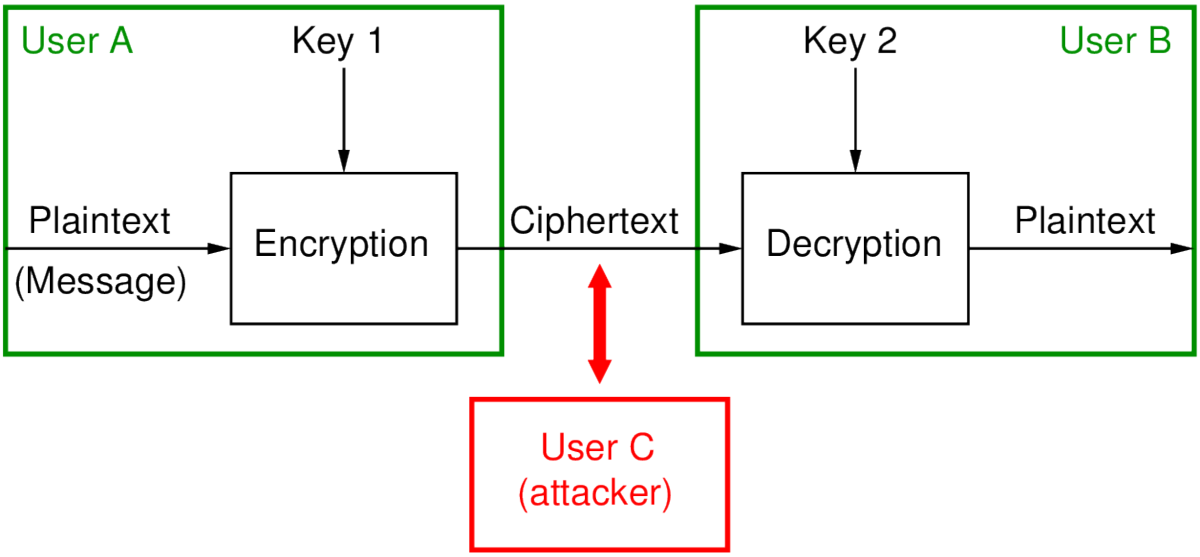


Figure 7.1: Model of Encryption for Confidentiality

A new form of cryptography was designed in the 1960's and 1970's, where the encrypter uses one key and the decrypter uses a different but related key. The keys are asymmetric. One of the keys is kept secret, while the other can be disclosed, i.e. made public.

We will focus on symmetric key ciphers initially, and return to public-key ciphers later.

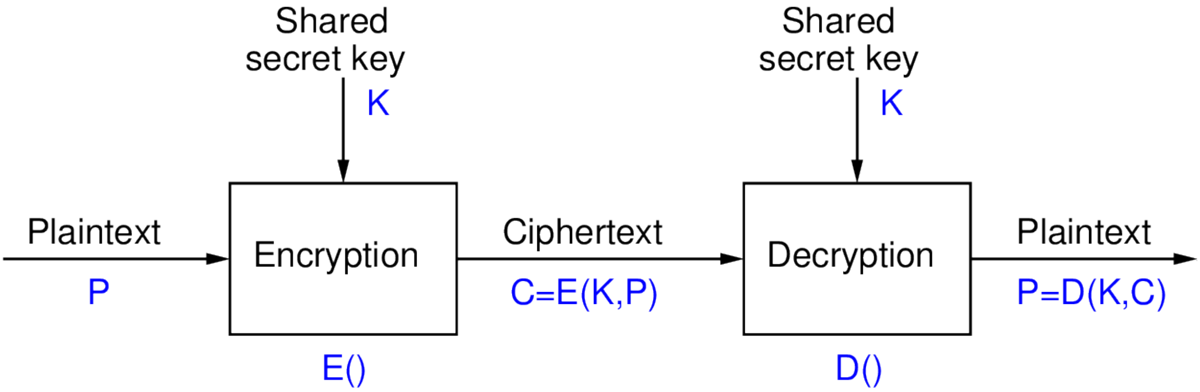


Figure 7.2: Symmetric Key Encryption for Confidentiality

We often use simple mathematical notation to describe the steps. E() is a function that takes two inputs: key K and plaintext P. It returns ciphertext C as output. E() represents the encryption algorithm. D() is the decryption algorithm.

Symmetric key encryption is the oldest form of encryption and involves both parties (e.g. sender and receiver) knowing the same secret key. Plaintext is encrypted with the secret key, and the ciphertext is decrypted with that secret key. If anyone else (i.e attacker) learns the secret key, then the system in not secure.

For symmetric key encryption to be secure, the algorithm must be well designed (strong, not easy to break) and the secret key must be kept secret. AES is an example of a strong algorithm, and it uses keys of length 128 bits or longer. One of the challenges of symmetric key encryption is informing the receiver of the secret key in advance: it must be done in a secure manner.

Substitution replace one element in plaintext with another

Permutation re-arrange elements (also called transposition)

Product systems multiple stages of substitutions and permutations, e.g. Feistel network, Substitution Permutation Network (SPN)

Symmetric key ciphers are designed around two basic operations: substitution and permutation. We have seen these operations when looking at classical ciphers. We also saw the principle that repeating the operations can make a cipher more secure. Modern ciphers are designed using these two basic operations, but repeated multiple times. For example, perform a substitution and then permutation, then repeat. The result is a "product system".

The Feistel network and SPN are two common design principles for modern ciphers and will be mentioned later when discussing block ciphers like AES and DES.

Block cipher process one block of elements at a time, typically 64 or 128 bits

Stream cipher process input elements continuously, e.g. 1 byte at a time, by XOR plaintext with keystream

Originally the idea was that block ciphers were suitable for processing large amounts of data when there were no strict time constraints. Stream ciphers were fast and suitable for real-time applications. For example, for encrypting real-time voice, as the data (plaintext) is generated, it needs to be quickly encrypted and then the ciphertext transmitted across a network. By encrypting only a small amount of plaintext at a time and using the extremely fast XOR operation, stream ciphers could perform the encryption without introducing significant delay.

However nowadays, the dedicated hardware support for block ciphers like AES, there is not a significant difference in performance (delay) of block and stream ciphers. Hence we see block ciphers (in particular, AES) used in scenarios for which stream ciphers were originally designed for.

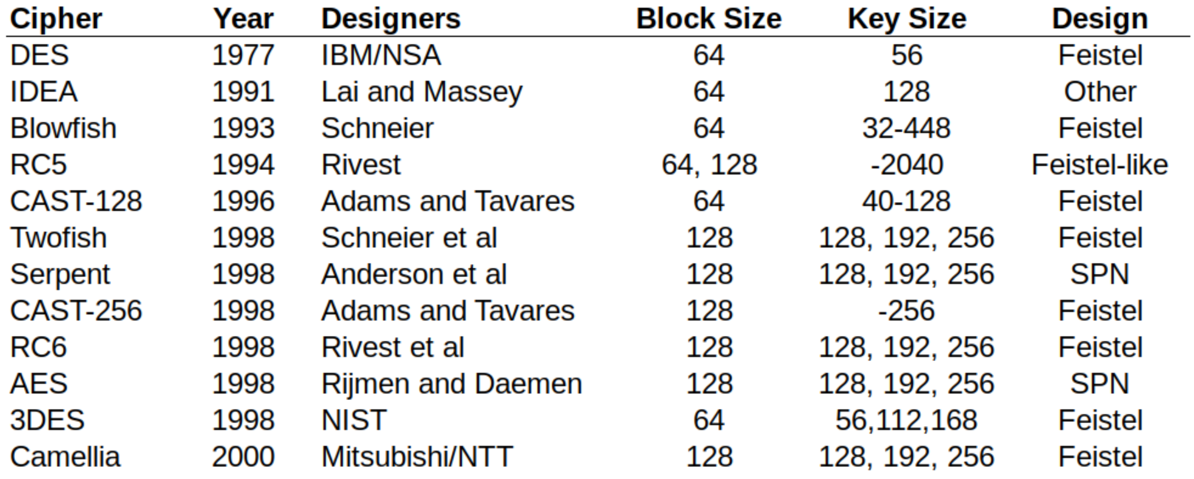
We will focus on block ciphers initially, and return to stream ciphers later.

Data Encryption Standard (DES) Became a US government standard in 1977 and widely used for more than 20 years; key is too short

Advanced Encryption Standard (AES) Standardised a replacement of DES in 1998, and now widely used. Highly recommended for use.

While no longer recommended or in widespread use, DES was the first cipher that saw widespread use. The primary limitation of DES however was the key was eventually subject to a brute force attack. It was only 56 bits.

While Triple DES, which used the original DES but expanded the key length, was popular for awhile, a new cipher was needed to perform well in a variety of hardware



Cipher	Year	Designers	Block Size	Key Size	Design
DES	1977	IBM/NSA	64	56	Feistel
IDEA	1991	Lai and Massey	64	128	Other
Blowfish	1993	Schneier	64	32-448	Feistel
RC5	1994	Rivest	64, 128	-2040	Feistel-like
CAST-128	1996	Adams and Tavares	64	40-128	Feistel
Twofish	1998	Schneier et al	128	128, 192, 256	Feistel
Serpent	1998	Anderson et al	128	128, 192, 256	SPN
CAST-256	1998	Adams and Tavares	128	-256	Feistel
RC6	1998	Rivest et al	128	128, 192, 256	Feistel
AES	1998	Rijmen and Daemen	128	128, 192, 256	SPN
3DES	1998	NIST	64	56,112,168	Feistel
Camellia	2000	Mitsubishi/NTT	128	128, 192, 256	Feistel

Figure 7.3: Common Symmetric Key Block Ciphers

platforms. AES was standardised in 1998 and continues to be the recommended symmetric key block cipher for most applications today. There are no known practical attacks that cannot be defended.

DES and AES are covered in depth later.

Figure 7.3 lists common symmetric key encryption block ciphers starting with DES, through to around the time of AES. Most block ciphers operate on blocks of 64 or 128 bits, and support a range of key lengths. There are three main design principles: Feistel network or structure, Substitution Permutation Network, or Lai-Massey.

AES is still highly recommended for most applications. There have been newer proposals since then, however very few are standards or see wide spread usage. A recent trend is on developing "lightweight" ciphers that perform well on very small devices, e.g. sensors.

A detailed review of block ciphers is Roberto Avanzi's "A Salad of Block Ciphers: The State of the Art in Block Ciphers and their Analysis", 2017, which is available for free at https://eprint.iacr.org/2016/1171.pdf

7.2 Attacks on Encryption

Cryptography, which is the study of cipher design, and cryptanalysis, i.e. breaking cipher designs, go hand-in-hand. Together these areas are study are cryptology. Let's now look from the attackers perspectives. Note that when we use the word "attacker", we don't necessarily mean a malicious entity. That is, we are not judging whether the entity performing the attack is good or evil.

Video
Attacks on Encryption (28 min; Mar 2020)
https://www.youtube.com/watch?v=yuiGyCx3WFA

7.2.1 Aims and Knowledge of the Attacker

First we list the general aims of an attacker, as well as assumptions we often make about the attacker.

- Study of ciphers and attacks on them is based on assumptions and requirements
  - Assumptions about what attacker knows and can do, e.g. intercept messages, modify messages
  - Requirements of the system/users, e.g. confidentiality, authentication
- Normally assumed attacker knows cipher
  - Keeping internals of algorithms secret is hard
  - Keeping which algorithm used secret is hard
- Attacker also knows the ciphertext
- Attacker has two general approaches
  - "Dumb": try all possible keys, i.e. brute force
  - "Smart": use knowledge of algorithm and ciphertext/plaintext to discover unknown information, i.e. cryptanalysis

7.2.2 Brute Force Attacks

Brute force is the "dumb" or naive approach an attacker can take. It involves trying keys until the correct plaintext is found.

Key	Key	Worst case time at speed:		
length	space	10^9/sec	10^{12}/sec	10^{15}/sec
32	2^{32}	4 sec	4 ms	4 us
56	2^{56}	833 days	20 hrs	72 sec
64	2^{64}	584 yrs	213 days	5 hrs
80	2^{80}	10^7 yrs	10^4 yrs	38 yrs
100	2^{100}	10^{13} yrs	10^{10} yrs	10^7 yrs
128	2^{128}	10^{22} yrs	10^{19} yrs	10^{16} yrs
192	2^{192}	10^{41} yrs	10^{38} yrs	10^{35} yrs
256	2^{256}	10^{60} yrs	10^{57} yrs	10^{54} yrs
26!	2^{88}	10^{10} yrs	10^7 yrs	10^4 yrs

Table 7.1: Worst Case Brute Force Time for Different Keys

Table 7.1 shows, for different key lengths, the time it takes to try every key if a single computer could make attempts at one of three rates: 10^9 per second, 10^{12} per second, or 10^{15} per second. There are not necessarily realistic speeds, although roughly represent lower and upper limits for today's computing power.

While this table presents the worst case time, in most cases, it is not much different from the average time. Recall the average time is about half of the worst case time. For a 128 bit key at 10^{15} decrypts per second, the worst case time is about 1×10^{16} years, and the average time is about 0.5×10^{16}. That is, both about 10^{16} years. With such large times, cutting the time in half makes no practical difference.

Note that the last line is for a key for a monoalphabetic English cipher. There are 26! possible keys which is equivalent to a binary key of about 88 bits.

For comparison, the age of the Earth is approximately 4×10^9 years and the age of the universe is approximately 1.3×10^{10} years.

7.2.3 Cryptanalysis

Cryptanalysis is the "smart" approach to breaking ciphers. The attacker uses knowledge of the ciphers, as well as expected patterns in ciphertext and plaintext to find unknown information (e.g. keys or plaintext).

Attacks on ciphers can be classified based on how much information an attacker is assumed to know to successfully perform the attack. We describe different classifications in the following.

1. Ciphertext Only Attack
2. Known Plaintext Attack
3. Chosen Plaintext Attack
4. Chosen Ciphertext Attack
5. Chosen Text Attack

We describe the different attacks in the following.

Ciphertext Only Attack

- Attacker knows:
  - encryption algorithm
  - ciphertext
- Hardest type of attack
- If cipher can be defeated by this, then cipher is weakest

The common assumption is that an attacker knows the encryption algorithm and ciphertext, and that they had no influence over the choice of ciphertext. This is referred to a *ciphertext only* attack. A cipher that is subject to a ciphertext only attack is the weakest of the groups of attacks we will consider.

However if a cipher cannot be defeated by a ciphertext only attack, then it still may be defeated if the attacker has additional information. The following defines these additional attacks. They all assume the attacker has the same information as a ciphertext only attack (i.e. encryption algorithm and ciphertext), but also make additional assumptions about other known information and the ability to select/influence values. Generally, the more information an attacker knows or can control, the easier their task of defeating a cipher.

Known Plaintext Attack

- Attacker knows:
  - encryption algorithm
  - ciphertext
  - one or more plaintext–ciphertext pairs formed with the secret key
- E.g. attacker has intercept past ciphertext *and* somehow discovered their corresponding plaintext
- All pairs encrypted with the same secret key (which is unknown to attacker)

In a Known Plaintext Attack (KPA), the attacker also has access to one or more pairs of plaintext/ciphertext. That is, assume the ciphertext known, C_{known}, was obtained using key $K_{unknown}$ and plaintext $P_{unknown}$ (either of which the attacker is trying to find). The attacker also knows at least C_1 *and* P_1, where C_1 is the output of encrypting P_1 with key $K_{unknown}$. That is, the attacker knows a pair (P_1, C_1). They may also know other pairs (obtained using the same key $K_{unknown}$).

How could an attacker known past plaintext/ciphertext pairs? A simple example is if the plaintext messages were only valid for a limited time, after which they become public. Such as coordinates for a public event to take place. Before the event takes place the coordinates are encrypted and secret. But after the event takes place, while the coordinates were decrypted, the attacker has learnt the value of the coordinates/plaintext (without knowing the key).

Generally, the more pairs of plaintext/ciphertext known, the easiest it is to defeat a cipher.

Chosen Plaintext Attack

- Attacker knows:
  - encryption algorithm
  - ciphertext
  - plaintext message chosen by attacker, together with its corresponding ciphertext generated with the secret key

In a Chosen Plaintext Attack (CPA) the attacker is able to select plaintexts to be encrypted and obtain their ciphertext (but not knowing the key used in the encryption). In such an attack, the attacker may select plaintext messages that have characteristics that make it easier to break the cipher. Ability to select plaintext and have it encrypted is common for public key ciphers (since the encryption key is public but the decryption key is private), which should be designed to be resistant to such attacks.

Chosen Ciphertext Attack

- Attacker knows:
  - encryption algorithm

  - ciphertext
  - ciphertext chosen by attacker, together with its corresponding decrypted plaintext generated with the secret key

- Attackers aim is to find the secret key (not the plaintext)

In a Chosen Ciphertext Attack (CCA) the attacker chooses a ciphertext, and obtains the corresponding plaintext, in an attempt to discover a secret key. Note in this attack, the aim is to find the secret key. If the attacker has a way to obtain plaintext from a chosen ciphertext, then they could simply intercept ciphertext to find plaintext. A CCA normally involves the attacker tricking a user to decrypt ciphertext and provide the plaintext.

There are variations of the above types of attacks, and the details of the attacks may be quite different, however this classification is sufficient to demonstrate that successful cryptanalysis depends partially on the amount of information known to the attacker.

7.2.4 Measuring Security

Is a cipher security? To answer such a question, methods of measuring security must be defined.

Unconditionally Secure Ciphertext does not contained enough information to derive plaintext or key

- One-time pad is only unconditionally secure cipher (but not very practical)

Computationally Secure If:

- cost of breaking cipher exceeds value of encrypted information
- or time required to break cipher exceeds useful lifetime of encrypted information
- Hard to estimate value/lifetime of some information
- Hard to estimate how much effort needed to break cipher

In theory we would like an unconditionally secure cipher. However in practice, we aim for computationally secure. Unfortunately it is difficult to measure if a cipher is computationally secure. For modern ciphers their security is judged based on the known theoretical and practical attacks (e.g. resistant to CCA or not) as well as the metrics in the following.

Time: usually measured as *number of operations*, since real time depends on implementation and computer specifics

- Operations are encrypts or decrypts; ignore other processing tasks
- E.g. worst case brute force of k-bit key takes 2^k (decrypt) operations

Amount of Memory: temporary data needed to be stored during attack

Known information: number of known plaintext/ciphertext values attacker needs to know in advance to perform attack

While time to break the cipher is the metric of interest, it is usually simplified to number of operations. For cryptanalysis, successful attacks should take fewer operations than brute force. That is, an attack that takes more operations the a brute force attack is considered an unsuccessful attack.

Often attacks requires intermediate values to be stored in memory while performing the attack. The less memory needed, the better the attack.

As seen in the previous classification, known plaintext, chosen plaintext and chosen ciphertext attacks all require the attacker to know additional information. The more information necessary for the attack to be successful, the poorer the attack is. For example, a known plaintext attack that will be successful if 1,000,000 pairs of plaintext/ciphertext are known, is better than a known plaintext attack that requires 2,000,000 pairs.

Video
Measuring Attacks on Ciphers (4 min; Mar 2021)
https://www.youtube.com/watch?v=3tfqACxHUSA

7.3 Block Cipher Design Principles

Block ciphers are the most common type of ciphers. They are designed to encrypt a single fixed length block of bits.

- Encrypt a block of plaintext as a whole to produce same sized ciphertext
- Typical block sizes are 64 or 128 bits
- Modes of operation used to apply block ciphers to larger plaintexts

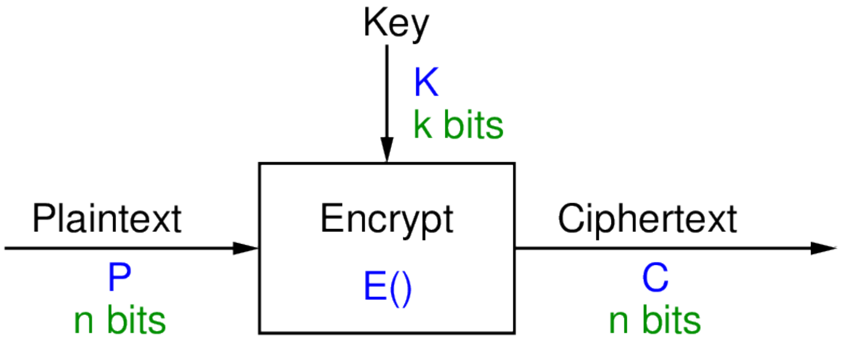
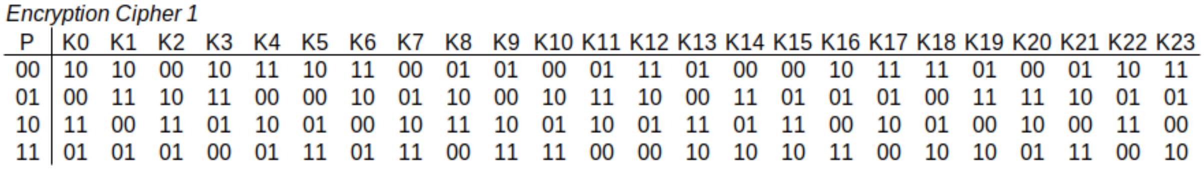


Figure 7.4: Block Cipher with n bit blocks

Modes of operation are covered in Chapter 11.

Video
Block Cipher Design Principles (9 min; Mar 2021)
https://www.youtube.com/watch?v=ULKX4Xqpclg

Let's look at some simple, ideal block ciphers to illustrate basic concepts, which will then lead to common design principles used to create block ciphers in use today. In its simplest form, a block cipher maps an n-bit plaintext block to a n-bit ciphertext block, with the exact mapping determined by the cipher design and selected key. The mapping can be viewed as a lookup table.



Encryption Cipher 1

P	K0	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15	K16	K17	K18	K19	K20	K21	K22	K23
00	10	10	00	10	11	10	11	00	01	01	00	01	11	01	00	00	10	11	11	01	00	01	10	11
01	00	11	10	11	00	00	10	01	10	00	10	11	10	00	11	01	01	01	00	11	11	10	01	01
10	11	00	11	01	10	01	00	10	11	10	01	10	01	11	01	11	00	10	01	00	10	00	11	00
11	01	01	01	00	01	11	01	11	00	11	11	00	00	10	10	10	11	00	10	10	01	11	00	10

Figure 7.5: Simple Ideal 2-bit Block Cipher 1

Figure 7.5 is an example of a 2-bit ideal block cipher. The table shows input plaintext blocks in the left column, different keys in the top row, and the resulting output ciphertext block in the body of the table. To be used for sending a confidential message, both the sender and receiver would know the table (e.g. stored in memory on their devices), or some way to calculate the table) and agree upon the key to use. For a given plaintext block, the sender looks up the key to find the output ciphertext to send. The receiver looks up the receiver ciphertext in the column of the key, and the row determines the plaintext.

Exercise 7.1 (Encrypt with Ideal Cipher 1)**.** Encrypt the message *Tokyo* using the above ideal 2-bit block cipher 1 with key K6.

Solution 7.1 (Encrypt with Ideal Cipher 1)**.** As the example block cipher operates on 2-bit binary blocks, but a five letter English message is to be encrypted we will make assumptions about the encoding and mode of operation to be used.

First, we will assume ASCII (or UTF-8) encoding is to be used (see Section B.1.4). Each letter will map to an 8-bit value, i.e. `T` = `01010100`, `o` = `01101111`, `k` = `01101011`, and `y` = `01111001`. The resulting plaintext in binary is 40 bits:

`0101010001101111011010110111100101101111`

Second, as we have 40 bits of plaintext, but a 2-bit block cipher, we will assume each 2-bit block of plaintext will be encrypted (20 blocks in total), and the resulting 20 ciphertext blocks will be concatenated to produce final 40 bit ciphertext. This naive approach is referred to as the Electronic Code Book mode of operation. Modes of operations are discussed in Chapter 11. The 20 plaintext blocks are:

`01 01 01 00 01 10 11 11 01 10 10 11 01 11 10 01 01 10 11 11`

Consider the 1st plaintext block of `01`, using key `K6`, looking up the block cipher table returns a ciphertext block of `10`. We know have the first of 20 ciphertext blocks and can move on to the 2nd ciphertext block. It turns out the next plaintext block is the same as the first (`01`), and since the same key is used (`K6`), the same ciphertext block will be output (10). In fact, the first three plaintext blocks are the same, so the ciphertext blocks so far are:

```
10 10 10
```

The 4th plaintext block is 00. Looking up in the table with key K6 produces output ciphertext block 11. We know have:

```
10 10 10 11
```

Continuing with all 20 plaintext blocks will produce ciphertext blocks:

```
10 10 10 11 10 00 01 01 10 00 00 01 10 01 00 10 10 00 01 01
```

Concatenating all ciphertext blocks together produces the ciphertext:

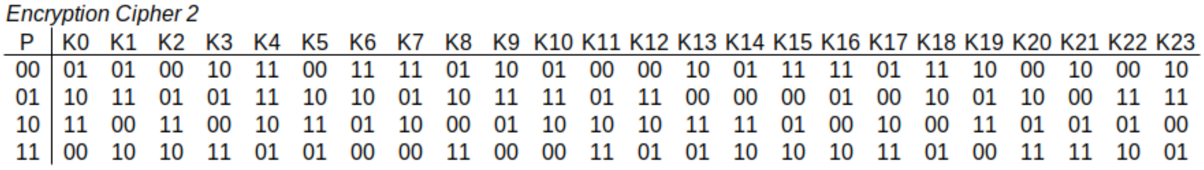
```
1010101110000101100000011001001010000101
```

Should the ciphertext be encoded as ASCII/UTF8 to complete the encryption? It could be, but note that some of the characters may note be printable (e.g. ESCape or ACK). For block ciphers we typically operate on binary plaintext and ciphertext. Encoding and decoding between binary and other formats is not normally part of the cipher, so we will leave the ciphertext as a sequence of bits.

The above exercise identified several issues that arise when applying an ideal block cipher:

- Encoding/decoding: independent of block cipher, which operate only in binary values
- Mode of operation: typically independent of block cipher, which operate only on a single block
- Repetition of plaintext blocks: undesirable. Make block size larger and use mode of operation that obscures repetition
- Key space: larger block size needed to allow more keys in ideal block cipher
- Implementing an ideal block cipher: how are they generated? can all values be stored?

The following questions will explore some of these issues further.



Encryption Cipher 2

P	K0	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15	K16	K17	K18	K19	K20	K21	K22	K23
00	01	01	00	10	11	00	11	11	01	10	01	00	00	10	01	11	11	01	11	10	00	10	00	10
01	10	11	01	01	11	10	10	01	10	11	11	01	11	00	00	00	01	00	10	01	10	00	11	11
10	11	00	11	00	10	11	01	10	00	01	10	10	10	11	11	01	00	10	00	11	01	01	01	00
11	00	10	10	11	01	01	00	00	11	00	00	11	01	01	10	10	10	11	01	00	11	11	10	01

Figure 7.6: Simple Ideal 2-bit Block Cipher 2

Figure 7.6 shows a different 2-bit ideal block cipher. It maps plaintext to ciphertext in a different order than cipher 1.

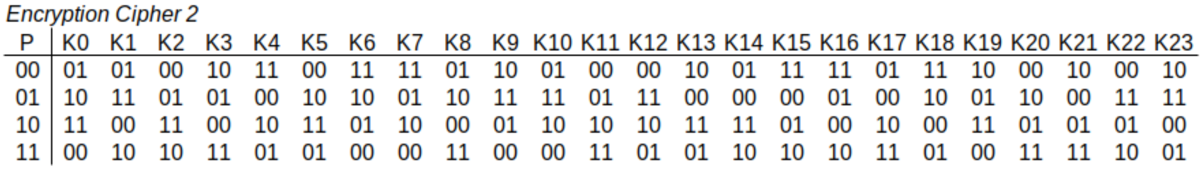
This example is just used for illustrative purposes. If you had an ideal block cipher that covered every permutation of plaintext values, then only a single cipher is needed.

Question 7.1 (What is plaintext with key K13, ciphertext 11 with ideal cipher 2?)**.** What is plaintext with key K13, ciphertext 11 with ideal cipher 2?

Decryption also involves a lookup. In the column for key K13, identify the ciphertext 11, and the row indicates the original plaintext 10.

Question 7.2 (What is plaintext with key K4, ciphertext 11 with ideal cipher 2?)**.** What is plaintext with key K4, ciphertext 11 with ideal cipher 2?

Same cipher, same ciphertext but different key. However in column of K4 there are two values of ciphertext 11. So we cannot determine for sure what was the original plaintext: 00 or 10. This actually is a trick question, since the cipher design is in error. A cipher must be reversible, so decryption is possible. This is an example of a cipher design error that includes an irreversible mapping.



Encryption Cipher 2

P	K0	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15	K16	K17	K18	K19	K20	K21	K22	K23
00	01	01	00	10	11	00	11	11	01	10	01	00	00	10	01	11	11	01	11	10	00	10	00	10
01	10	11	01	01	00	10	10	01	10	11	11	01	11	00	00	00	01	00	10	01	10	00	11	11
10	11	00	11	00	10	11	01	10	00	01	10	10	10	11	11	01	00	10	00	11	01	01	01	00
11	00	10	10	11	01	01	00	00	11	00	00	11	01	01	10	10	10	11	01	00	11	11	10	01

Figure 7.7: Simple Ideal 2-bit Block Cipher 2 (fixed)

Figure 7.7 shows the fixed cipher: it is now reversible, and decryption is possible for all values of key and ciphertext.

Question 7.3 (How many bits are needed to represent the key in cipher 2?)**.** The example 2-bit ideal block cipher 2 (as well as cipher 1) list 24 different keys (or mappings from plaintext to ciphertext). How many bits are needed to represent a key for this cipher?

Firstly, why are 24 keys listed? With a 2-bit block, there are $2^2 = 4$ possible blocks, i.e. 00, 01, 10, and 11. There are $4! = 24$ different ways to arrange those 4 plaintext blocks to produce ciphertext, i.e. 24 permutations of the plaintext blocks. A key is used to select the distinct permutation.

With key length of 1 bit, we can represent $2^1 = 2$ possible keys. With a key length of 2 bits, we can represent $2^2 = 4$ possible keys. With a key length of 3 bits, we can represent $2^3 = 8$ possible keys. With a key length of 4 bits, we can represent $2^4 = 16$ possible keys. With a key length of 5 bits, we can represent $2^5 = 32$ possible keys. That is, a key length of 4 bits is not enough to represent our 24 keys, but a key length of 5 is. Therefore we need a 5-bit key for this ideal 2-bit block cipher.

Question 7.4 (How to reduce repetition of plaintext blocks?)**.** With a 2-bit ideal block cipher, with a long plaintext, many of plaintext blocks will repeat. This is bad for security (see Modes of Operation). What can you change in the design of an ideal block cipher that reduces repetition of plaintext blocks?

Increasing the block size for a block cipher will reduce the change of block repetition. Recall the first example of the 2-bit ideal block cipher encrypting *Tokyo*. The plaintext was 40-bits, resulting in 20 blocks. As there are only $2^2 = 4$ different plaintext values, there will be repetition. On average (if the plaintext was random, which is not likely but it simplifies the analysis), each plaintext value will be repeated $20/4 = 5$ times.

If however a 3-bit ideal block cipher was used, there would be $2^3 = 8$ different plaintext values. There would be 14 blocks (40/3, with the last block having just 1 bit of plaintext). On average, each plaintext value will be repeated 14/8, which is less than 2 times.

Increasing to a 4-bit ideal block cipher gives 16 different plaintext values, 10 blocks, and a possibility there will be no repetition. Of course if the plaintext is much longer than 40 bits, then repetition is still likely.

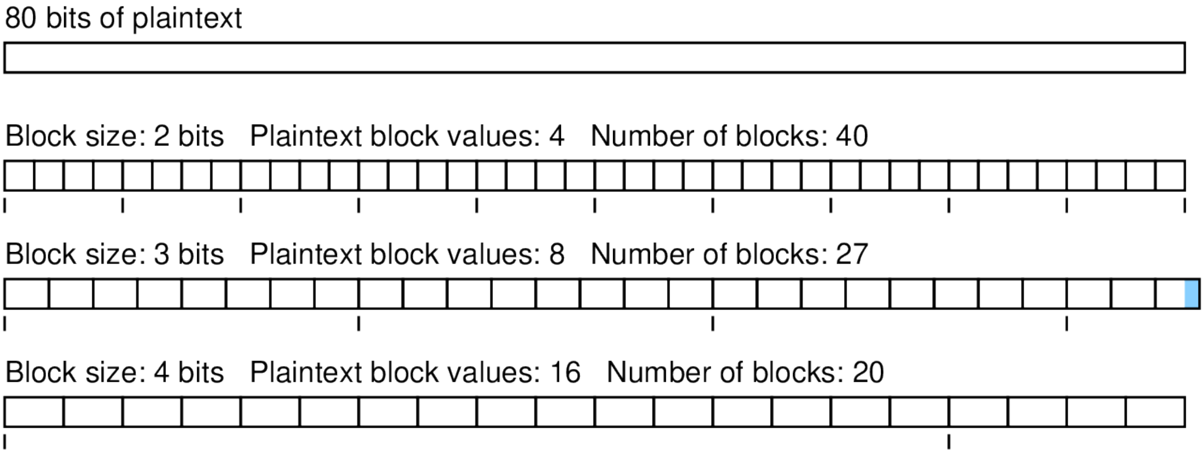

80 bits of plaintext
Block size: 2 bits Plaintext block values: 4 Number of blocks: 40
Block size: 3 bits Plaintext block values: 8 Number of blocks: 27
Block size: 4 bits Plaintext block values: 16 Number of blocks: 20

Figure 7.8: Impact of Block Sizes for 80 bit Plaintext

Figure 7.8 illustrates the impact of different block sizes for an example 80 bit plaintext (whereas the previous example was a 40 bit plaintext).

Note that with a block size of 3 bits, the last block contains 2 bits of plaintext and 1 bit of *padding*. Padding is needed as all blocks must be the same size (since block ciphers operate on fixed sized blocks). There are different schemes for padding, e.g. bit padding, zero padding and PKCS7.

- n-bit block cipher takes n bit plaintext and produces n bit ciphertext
- 2^n possible different plaintext blocks
- Encryption must be reversible (decryption possible)
- Number of permutations of plaintext (and number of keys) is $2^n!$
- Design trade-offs:
  - Large block size to reduce plaintext repetitions (64-bits is good)
  - Key space large enough to avoid brute force, but small enough to make distribution practical
  - Small block size to simplify implementation

The trade-offs are conflicting, meaning ideal block ciphers are good in theory, but in practice we need a different design approach.

Exercise 7.2 (Ideal 64-bit Block Cipher)**.** Consider an ideal 64-bit block cipher. How many different different keys are possible? How many bits are needed to store a single key? How much space is required to store the mappings?

Solution 7.2 (Ideal 64-bit Block Cipher)**.** We will not attempt to list all keys. With 64-bit blocks, there are $2^{64}!$ different permutations or mappings, meaning $2^{64}!$ possible keys. To store a single key, we need about $\log_2(2^{64}!)$ bits. Our software calculator will not handle this, not even `bc`. So let's try Wolfram Alpha, which returns 1.15398×10^{21}. That means about 10^{21} bits are needed to store a key. That is approximately 125,000,000 TB. If someone wanted to send a short encrypted message to you, they would first need to exchange a 125,000,000 TB key with you. Hence we see an ideal block cipher with large blocks is not practical due to the key length.

For storage of the mappings, consider if you had to create a table similar to the 2-bit ideal block ciphers. There are $2^{64}!$ columns, representing the keys. There are 2^{64} rows, representing the possible plaintext values. Each cell in the table contains a 64-bit, or 8 Byte, ciphertext value. So the storage space needed is $2^{64}! \times 2^{64} \times 8$ Bytes. If you attempt to calculate this you will quickly see it is not practical to store the entire table.

Video
Ideal Block Cipher (8 min; Mar 2021)
https://www.youtube.com/watch?v=-LjOIGjURGs

To overcome the limitations of ideal block ciphers, Horst Feistel designed a general scheme that is practical in the sense of implementation and key lengths, but still achieves suitable security.

- Ideal block ciphers are not practical
- Feistel proposed applying two or more simple ciphers in sequence so final result is cryptographically stronger than component ciphers
- n-bit block length; k-bit key length; 2^k transformations
- Feistel cipher alternates: substitutions, transpositions (permutations)
- Applies concepts of diffusion and confusion
- Applied in many ciphers today
- Approach:
  - Plaintext split into halves
  - Subkeys (or round keys) generated from key
  - Round function, F, applied to right half
  - Apply substitution on left half using XOR
  - Apply permutation: interchange to halves

For example, with a 64-bit block cipher, there are 2^{64} possible mappings/keys, meaning the key length is $\log_2(2^{64}) = 64$ bits.

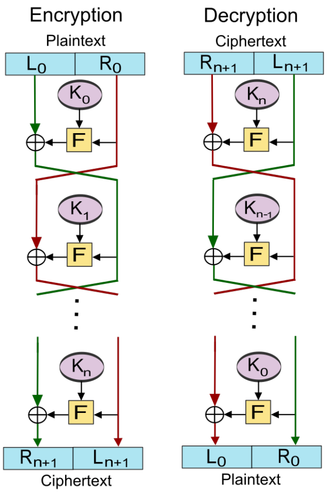
- Diffusion

  - Statistical nature of plaintext is reduced in ciphertext
  - E.g. A plaintext letter affects the value of many ciphertext letters
  - How: repeatedly apply permutation (transposition) to data, and then apply function

- Confusion

  - Make relationship between ciphertext and key as complex as possible
  - Even if attacker can find some statistical characteristics of ciphertext, still hard to find key
  - How: apply complex (non-linear) substitution algorithm

Diffusion and confusion are concepts introduced by Claude Shannon. See a summary of Shannon's contributions in telecommunications, digital circuits and cryptography in Chapter C.



Credit: Amirki, https://commons.wikimedia.org/wiki/File:Feistel_cipher_diagram_en.svg, CC BY-SA 3.0

Figure 7.9: Feistel Encryption and Decryption

You don't need to know the details of the Feistel structure. Just be aware that it is a design principle used in many block ciphers, including DES.

- Exact implementation depends on various design features

  - Block size, e.g. 64, 128 bits: larger values leads to more diffusion
  - Key size, e.g. 128 bits: larger values leads to more confusion, resistance against brute force
  - Number of rounds, e.g. 16 rounds
  - Subkey generation algorithm: should be complex
  - Round function F: should be complex

- Other factors include fast encryption in software and ease of analysis

- Trade-off: security vs performance

Video
Ideal Block Cipher vs Feistel Structure (2 min; Mar 2021)
https://www.youtube.com/watch?v=5-zWE7GjjaQ

7.4 Stream Cipher Design Principles

Stream ciphers were designed to be fast using an XOR operation, and usually encrypt a bit or byte at a time.

- Encrypts a digital data stream one bit or one byte at a time
- One time pad is example; but practical limitations
- Typical approach for stream cipher:
    - Key (K) used as input to bit-stream generator algorithm
    - Algorithm generates cryptographic bit stream (k_i) used to encrypt plaintext
    - k_i is XORed with each byte of plaintext P_i
    - Users share a key; use it to generate keystream

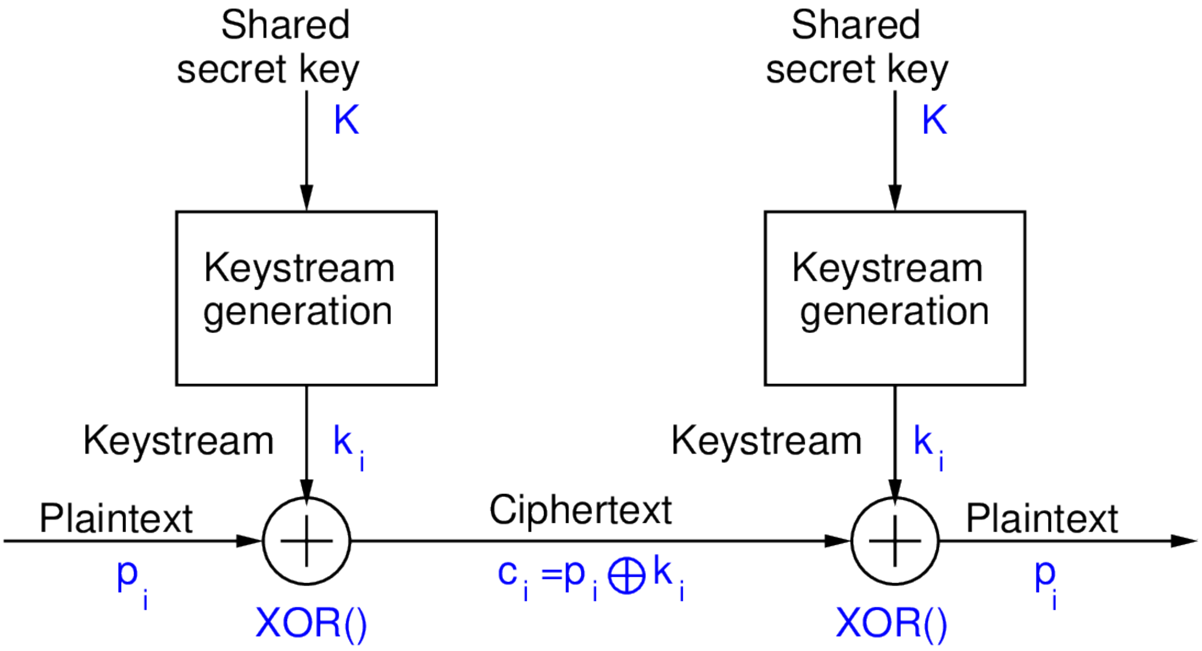


Figure 7.10: Stream Cipher Encrypt and Decrypt

Figure 7.10 illustrates the general operation of a stream cipher encryption and decryption. The sender uses a shared secret key K and an algorithm to generate effectively a random stream of bits. This random stream of bits is XORed with the plaintext bits as needed.

The receiver uses the same key and algorithm, which in turn generates the same random stream of bits. When XORed with the ciphertext, the original plaintext is output.

An issue when using stream ciphers is that a key cannot be re-used. This is usually addressed by introducing an initialisation value or vector (IV).

- Encrypting two different plaintexts with the same key leads to key re-use attack
    - Attacker intercepts two ciphertexts: $C_1 = P_1 \oplus k_1$ and $C_2 = P_2 \oplus k_1$
    - Properties of XOR: commutative and $A \oplus A = 0$
    - Attacker performs XOR on two ciphertexts
    - $C_1 \oplus C_2 = P_1 \oplus k_1 \oplus P_2 \oplus k_1 = P_1 \oplus P_2$
    - Even without knowing P_1 or P_2, attacker can easily use frequency analysis to discover both
- Solution: Use additional IV that changes for every encryption

Question 7.5 (When can key re-use attack be successful if IV is used?)**.** If a stream cipher is using a n-bit Initialisation Vector/Value (IV), but the same key, under what conditions is a key re-use attack possible? Assume the IV increments every time an encrypt operation is performed.

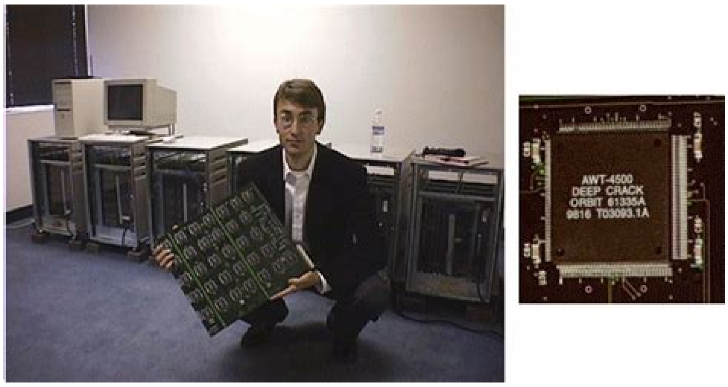
7.5 Example: Brute Force on DES

Now let's consider an example of brute force on a real cipher, DES.

- DES is 64-bit block cipher with 56-bit (effective) key length
- Developed in 1977, recommended standard until 1990's
- Brute force: 2^{56} operations
- Hardware built to perform brute force attack
    - 1998: DeepCrack
    - 2006: COPACABANA

In 1998, the Electronics Frontiers Foundation (EFF) developed DeepCrack to demonstrate how insecure DES was.

- Developed by EFF
- Cost less than $US250,000
- 80×10^9 keys/sec
- Solved DES challenge in 56 hours



Credit: Wikimedia, CC0 1.0 Public Domain https://commons.wikimedia.org/wiki/File:Paul_kocher_deepcrack.jpg

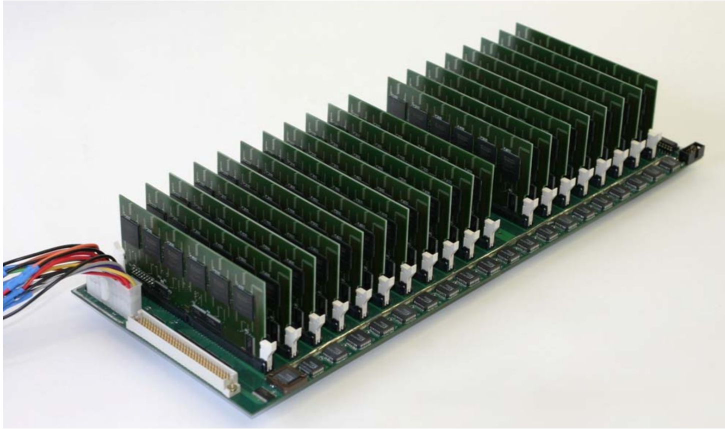
Figure 7.11: Paul Kocher and DeepCrack

- See www.cryptography.com and www.eff.org

Video
DeepCrack Brute Force on DES (1 min; Mar 2021)
https://www.youtube.com/watch?v=SIH2lgfV3Ao

In 2006, as a demonstration of their hardware, SciEngines developed COPACABANA.

- Joint effort by SciEngines and German universities
- 120 Field Programmable Gate Arrays (FPGAs), 400×10^6 keys/sec/FPGA
- For comparison, a Pentium 4: 2×10^6 keys/sec
- Brute force DES in 8.6 days
- Cost about $US10,000
- See www.sciengines.com



Credit: Copyright SciEngines GMBH

Figure 7.12: COPACABANA by SciEngines, 2006

Using the above example, we can roughly estimate what it would cost today to brute force DES.

- Moore's law: computers double speed every 1.5 years
- Alternative: computers halve in cost every 1.5 years
- $US10,000 to brute force DES in 2006
- Cost has halved about 10 times
- Cost to brute force DES in 2020: $10

A simplification of Moore's law is that computers double their speed every 1.5 years. In practice it is not that simple, but it is a useful rule to estimate the cost of brute force today. It means in 1.5 years time, you could buy a computer that double the speed if a new computer today, and at the same cost. Alternatively, you could buy a lower specced computer, which is the same speed as a new computer today, buy half the cost of today's computer.

Assuming computers halve in cost every 1.5 years, between 2006 and 2020 is 14 years. Over 15 years, there are 10 1.5 year periods, so the cost would halve 10 times. (Again since this is an estimate, let's use 15 years instead of 14). If you half $10,000 10 times, you get $9.76. That is, a $10 computer today can brute force DES in 8.6 days.

As brute force attacks can be parallelised easily, you could spend $100 on 10 computers (or buy a $100 computer) and break DES in less than a day. DES is not secure against a brute force attack (and hasn't been for a long time).

Video
SciEngines Copacabana Brute Force on DES (3 min; Mar 2021)
https://www.youtube.com/watch?v=8RGD7ckwoBI

7.6 Example: Brute Force on AES

Another demonstration of hardware brute force capabilities was again given by SciEngines in 2013, this time attacking AES.

- Rivyera S3 supported up to 128 Xilinx Spartan-3 FPGAs
- Approx $100 per FPGA (XCS5000)
- AES-128 Brute Force
  - 500×10^6 keys per sec
  - 4×10^6 keys per mW
- Biclique Attack
  - 945×10^6 keys per sec

Credit: Copyright SciEngines GMBH

Figure 7.13: RIVYERA S3-5000 by SciEngines, 2013

  - 7.3×10^6 keys per mW

FPGAs are essentially computer processors programmed for a specific task, in this case, decrypting with AES very fast. For about $12,800 a RIVYERA could decrypt AES-128 at a rate of 500×10^6 keys per second.

A known plaintext attack on AES is called the Biclique attack. The RIVYERA implementation of the Biclique attack could decrypted AES-128 at a rate of 945×10^6 keys per sec, about twice that of a brute force.

Now let's consider what it would take to break AES.

- AES-128 has key space of 2^{128}
- 2013: $US12,800 for 5×10^8 k/s
- Assume: computers double speed every 1.5 years
- 2020: Increase by $2^5 = 32$; 1.6×10^{10} k/s
  - $12,800: 6.7×10^{20} years
  - $12,800,000: 6.7×10^{17} years
  - $12,800,000,000: 6.7×10^{14} years
- Biclique attack about 2 to 4 times faster, but requires 2^{88} known plaintext/ciphertext pairs
- In 2035, cost $12,800,000,000 to brute force AES-128 in 670,000,000,000 years

Applying the same logic from analysis of DES brute force and Moore's law (i.e. every 1.5 years halve cost or double speed), we can perform a rough analysis of the cost/time to break AES-128. The numbers (dollars, years) are so large such that even if the approximations are incorrect by a factor of 1,000,000,000 (e.g. reducing 10^{14} years to $100,000$ years, then it is still impossible to break AES-128.

Video
SciEngines Rivyera Attack on AES (4 min; Mar 2021)
https://www.youtube.com/watch?v=KC3Z3yp0s5k

7.7 Example: Meet-in-the-Middle Attack

One way to increase the key length of a block cipher is to apply the cipher multiple times, each time using a different key. Applying the cipher twice is referred to as *double encryption*.

- Encrypt plaintext with one key, then encrypt output with another key

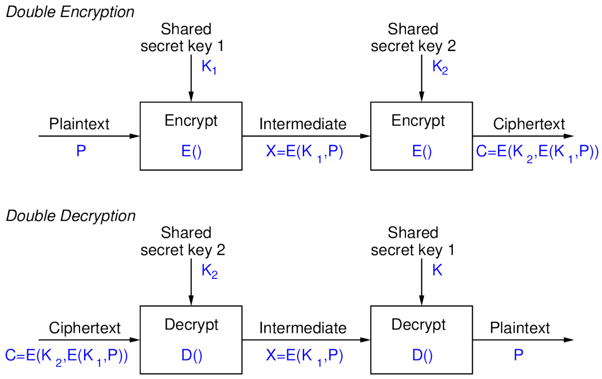


Figure 7.14: Double Encryption Concept

- Advantage: doubles the key length
  - Single version of cipher has k-bit key
  - Double version of cipher uses two different k-bit keys
  - Worst case brute force: 2^{2k}
- Advantage: uses an existing cipher
- Disadvantage: doubles the processing time
- Problem: double encryption is subject to *meet-in-the-middle* attack

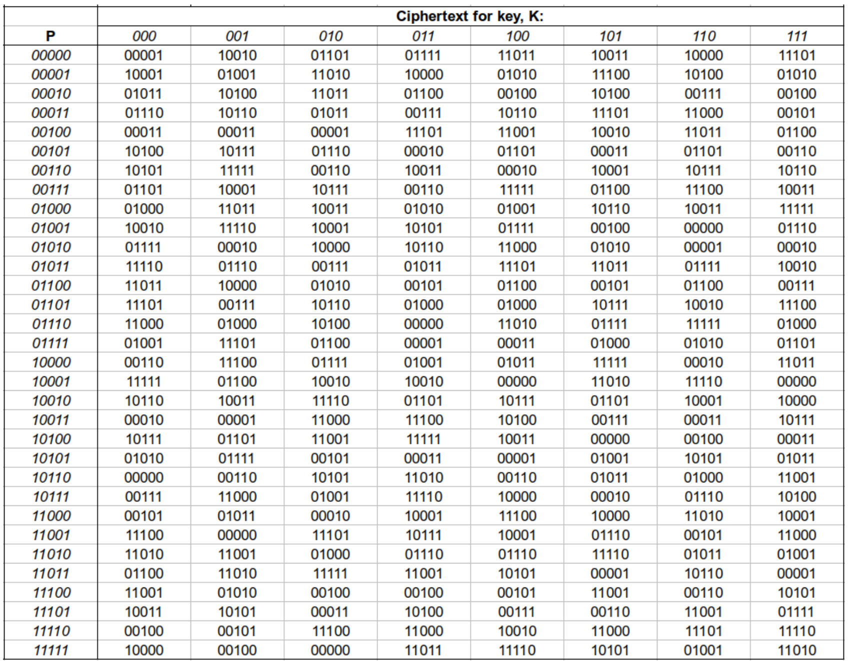
Double encryption was a (naive) option for extending the key length of DES. It effectively would double the key length from 56 bits to 112 bits. A new cipher would not have to be designed or analysed, and existing software/hardware implementations could be used.

But a meet-in-the-middle attack makes Double-DES (or double encryption on any block cipher) insecure.

- Double Encryption where key K is k-bits: $C = \mathrm{E}(K_2, \mathrm{E}(K_1, P))$
- Say $X = \mathrm{E}(K_1, P) = \mathrm{D}(K_2, C)$
- Attacker knows two plaintext, ciphertext pairs (P_a, C_a) and (P_b, C_b)
  1. Encrypt P_a using all 2^k values of K_1 to get multiple values of X
  2. Store results in table and sort by X
  3. Decrypt C_a using all 2^k values of K_2

4. As each decryption result produced, check against table
5. If match, check current K_1, K_2 on C_b. If P_b obtained, then accept the keys

- With two known plaintext, ciphertext pairs, probability of successful attack is almost 1
- Encrypt/decrypt operations required: $\approx 2 \times 2^k$ (twice as many as single encryption)



	Ciphertext for key, K:							
P	000	001	010	011	100	101	110	111
00000	00001	10010	01101	01111	11011	10011	10000	11101
00001	10001	01001	11010	10000	01010	11100	10100	01010
00010	01011	10100	11011	01100	00100	10100	00111	00100
00011	01110	10110	01011	00111	10110	11101	11000	00101
00100	00011	00011	00001	11101	11001	10010	11011	01100
00101	10100	10111	01110	00010	01101	00011	01101	00110
00110	10101	11111	00110	10011	00010	10001	10111	10110
00111	01101	10001	10111	00110	11111	01100	11100	10011
01000	01000	11011	10011	01010	01001	10110	10011	11111
01001	10010	11110	10001	10101	01111	00100	00000	01110
01010	01111	00010	10000	10110	11000	01010	00001	00010
01011	11110	01110	00111	01011	11101	11011	01111	10010
01100	11011	10000	01010	00101	01100	00101	01100	00111
01101	11101	00111	10110	01000	01000	10111	10010	11100
01110	11000	01000	10100	00000	11010	01111	11111	01000
01111	01001	11101	01100	00001	00011	01000	01010	01101
10000	00110	11100	01111	01001	01011	11111	00010	11011
10001	11111	01100	10010	10010	00000	11010	11110	00000
10010	10110	10011	11110	01101	10111	01101	10001	10000
10011	00010	00001	11000	11100	10100	00111	00011	10111
10100	10111	01101	11001	11111	10011	00000	00100	00011
10101	01010	01111	00101	00011	00001	01001	10101	01011
10110	00000	00110	10101	11010	00110	01011	01000	11001
10111	00111	11000	01001	11110	10000	00010	01110	10100
11000	00101	01011	00010	10001	11100	10000	11010	10001
11001	11100	00000	11101	10111	10001	01110	00101	11000
11010	11010	11001	01000	01110	01110	11110	01011	01001
11011	01100	11010	11111	11001	10101	00001	10110	00001
11100	11001	01010	00100	00100	00101	11001	00110	10101
11101	10011	10101	00011	10100	00111	00110	11001	01111
11110	00100	00101	11100	11000	10010	11000	11101	11110
11111	10000	00100	00000	11011	11110	10101	01001	11010

Figure 7.15: Example 5-bit Block Cipher

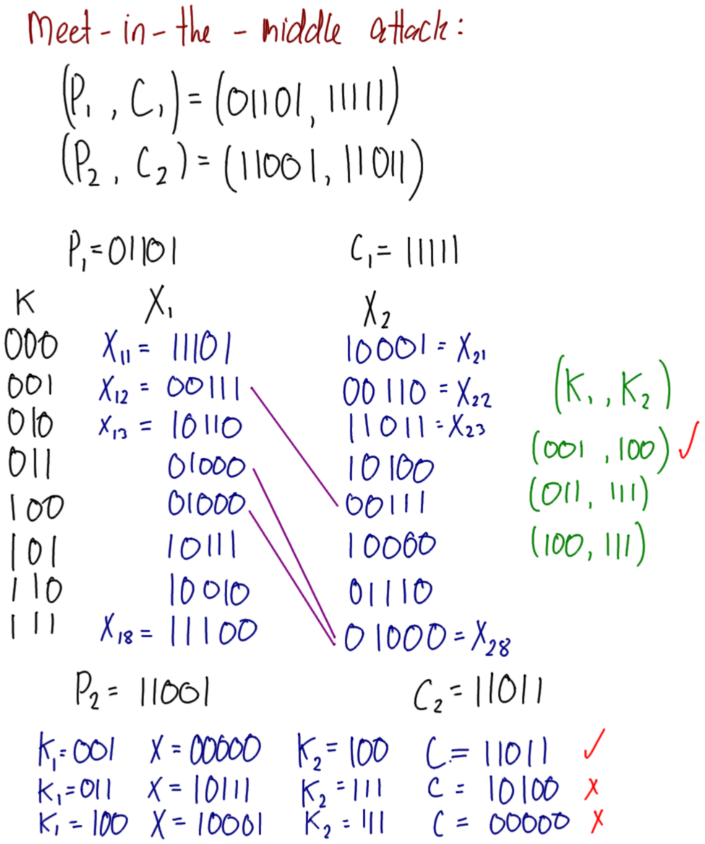
Figure 7.15 shows an example 5-bit block cipher with a 3-bit key. To encrypt, look in the left column to find the row of the plaintext, then look for the column corresponding to the key. The intersection of row and column gives the ciphertext.

This example block cipher is used in the Meet-in-the-Middle attack exercise.

Exercise 7.3 (Meet-in-the-Middle Attack)**.** Figure 7.15 shows an example 5-bit block cipher, referred to as *Bob's Cipher*. A double version of Bob's cipher, called *Double-Bob*, was used by two users to exchange multiple encrypted messages using the same 6-bit secret key. You have obtained the plaintext/ciphertext pairs of two of those messages: $(P_1, C_1) = (01101, 11111)$ and $(P_2, C_2) = (11001, 11011)$. Using a meet-in-the-middle attack, find the secret key.

Solution 7.3 (Meet-in-the-Middle Attack)**.** Figure 7.16 shows notes on performing the attack. Figure 7.17 shows calculations of the performance of the attack, and compares to an attack on Double-DES.

Video
Meet-in-the-Middle attack on 5-bit block cipher (52 min; Feb 2016)
https://www.youtube.com/watch?v=AwNlaN1w9jg



Meet-in-the-middle attack:

$(P_1, C_1) = (01101, 11111)$

$(P_2, C_2) = (11001, 11011)$

$P_1 = 01101$ $C_1 = 11111$

K	X_1	X_2
000	$X_{11} = 11101$	$10001 = X_{21}$
001	$X_{12} = 00111$	$00110 = X_{22}$
010	$X_{13} = 10110$	$11011 = X_{23}$
011	01000	10100
100	01000	00111
101	10111	10000
110	10010	01110
111	$X_{18} = 11100$	$01000 = X_{28}$

(K_1, K_2)

(001, 100) ✓

(011, 111)

(100, 111)

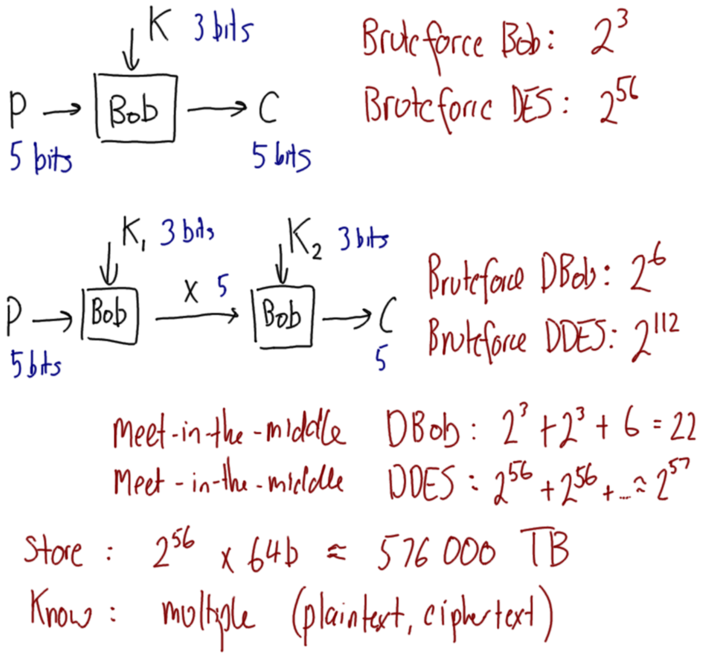
$P_2 = 11001$ $C_2 = 11011$

$K_1 = 001$ $X = 00000$ $K_2 = 100$ $C = 11011$ ✓

$K_1 = 011$ $X = 10111$ $K_2 = 111$ $C = 10100$ ✗

$K_1 = 100$ $X = 10001$ $K_2 = 111$ $C = 00000$ ✗

Figure 7.16: Solution for Meet-in-the-Middle Attack



Brute force Bob: 2^3

Brute force DES: 2^{56}

Brute force DBob: 2^6

Brute force DDES: 2^{112}

Meet-in-the-middle DBob: $2^3 + 2^3 + 6 = 22$

Meet-in-the-middle DDES: $2^{56} + 2^{56} + \ldots \approx 2^{57}$

Store: $2^{56} \times 64b \approx 576\,000$ TB

Know: multiple (plaintext, ciphertext)

Figure 7.17: Performance for Meet-in-the-Middle Attack

- Different variations:
  - Use 2 keys, e.g. Triple-DES 112 bits
  - Use 3 keys, e.g. Triple-DES 168 bits

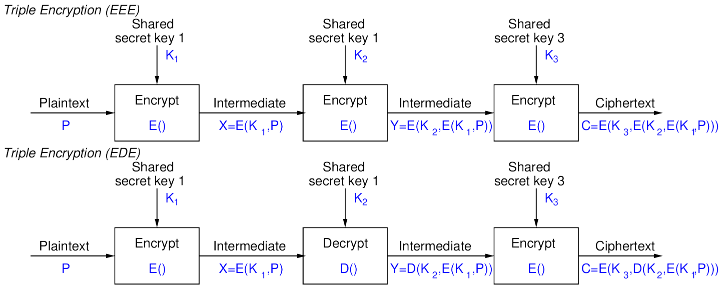


Figure 7.18: Triple Encryption Concept

- Why E-D-E? To be compatible with single DES:

$$C = \mathrm{E}(K_1, \mathrm{D}(K_1, \mathrm{E}(K_1, P))) = \mathrm{E}(K_1, P)$$

- Problem: 3 times slower than single DES

Figure 7.18 shows the concept of Triple Encryption, where two different keys are used. This effectively doubles the key strength compared to the original cipher. Another variation (not shown) would be to use three different keys, effectively tripling the key strength.

Note that if you use the same key for each step, then because of the E-D-E approach, this reverts to the original cipher. That is, if you use Triple-DES but use the same key in each step, this reverts to (single) DES. The benefit of this is that you can have an implementation of Triple-DES (which is built on the implementations of DES), and allow the user to choose a key to suit their needs: 1 key for DES, 2 keys for 112-bit security, 3 keys for 168-bit security.

7.8 Example: Cryptanalysis on Triple-DES and AES

Table 7.2 compares cryptanalysis on Triple-DES and AES against brute force attacks.

Video
Theoretical Attacks on DES and AES (2 min; Mar 2021)
https://www.youtube.com/watch?v=H001PSmfgMc

Cipher	Method	Key	Required resources:		
		space	Time	Memory	Known data
DES	Brute force	2^{56}	2^{56}	-	-
3DES	MITM	2^{168}	2^{111}	2^{56}	2^{2}
3DES	Lucks	2^{168}	2^{113}	2^{88}	2^{32}
AES 128	Biclique	2^{128}	$2^{126.1}$	2^{8}	2^{88}
AES 256	Biclique	2^{256}	$2^{254.4}$	2^{8}	2^{40}

- Known data: chosen pairs of (plaintext, ciphertext)
- Lucks: S. Lucks, Attacking Triple Encryption, in *Fast Software Encryption*, Springer, 1998
- Biclique: Bogdanov, Khovratovich and Rechberger, Biclique Cryptanalysis of the Full AES, in *ASIACRYPT2011*, Springer, 2011

Table 7.2: Cryptanalysis of Triple-DES and AES

Chapter 8

Data Encryption Standard

This chapter provides details of Data Encryption Standard (DES), with concepts demonstrated via a simplified, educational version called Simplified-DES. Many of the details serve mainly as reference, with little discussion.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

8.1 Overview of the Data Encryption Standard (DES)

- Symmetric block cipher
- 56-bit key, 64-bit input block, 64-bit output block
- Developed in 1977 by National Institute of Standards and Technology (NIST); designed by IBM (Lucifer) with input from National Security Agency (NSA)
- Principles used in other ciphers, e.g. 3DES, IDEA

8.2 Simplified-DES

To understand the details of a cipher, it often helps if you can perform the encryption (or decryption) steps yourself. However as common block ciphers operate on blocks of 64 bits or larger, and use similar sized keys, it is difficult to manually and efficiently perform operations. Therefore, to illustrate the principles of selected real ciphers, simplified versions have been developed. This section presents Simplified Data Encryption Standard (S-DES), which is a cut-down version of DES. For example, S-DES uses operates on 8-bit blocks, uses an 8-bit key and has only 2 rounds. As it is designed using the same principles as (real) DES but using smaller values, it is possible to step through an example encryption by hand. For some this can be a powerful way to understand the operations used in real DES. It is important however to note that S-DES is just for education; it is not a real cipher used in practice today or in the past. You will only find it referred to in textbooks and university classes.

- Input (plaintext) block: 8-bits

File: crypto/des.tex, r1966

- Output (ciphertext) block: 8-bits
- Key: 10-bits
- Rounds: 2
- Round keys generated using permutations and left shifts
- Encryption: initial permutation, round function, switch halves
- Decryption: Same as encryption, except round keys used in opposite order

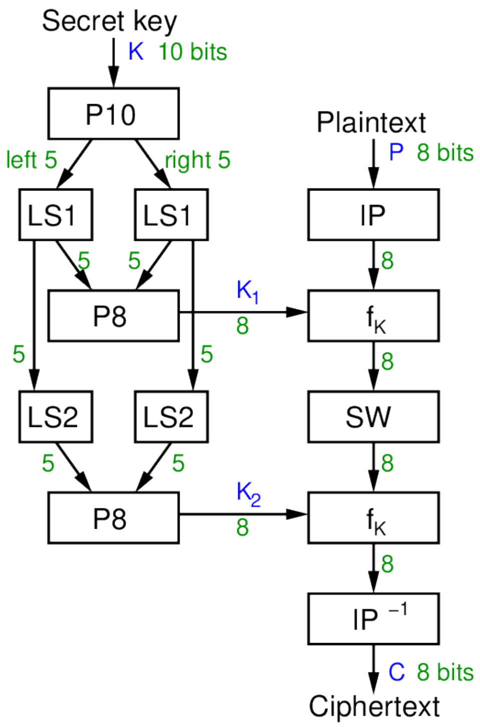


Figure 8.1: S-DES Key Generation and Encryption

Figure 8.1 shows the key generation and encryption steps of S-DES. Key generation, shown on the left, is used to generate round keys and is the same algorithm when used for both encryption and decryption. That is, the encrypter and decrypter will generate the exact same round keys.

The encrypter started with a shared secret key 10 bits long and 8 bits of plaintext. Two sub-keys, or round keys, K_1 and K_2 are generated using the key generation steps, which involve Permutations and Left Shifts.

Encryption applies an Initial Permutation, then a round function f_k (with details to be shown shortly), SWaps the two halves of the 8 bit output, then reapplies the round function, but using the 2nd round key as input. Encryption ends with the inverse of the Initial Permutation.

Figure 8.2 shows the key generation and decryption. Decryption is in fact identical to encryption, except the round keys are used in the opposite order. That is, for encryption round key K_1 is used first, then round key K_2. For decryption, K_2 is used first and then K_1.

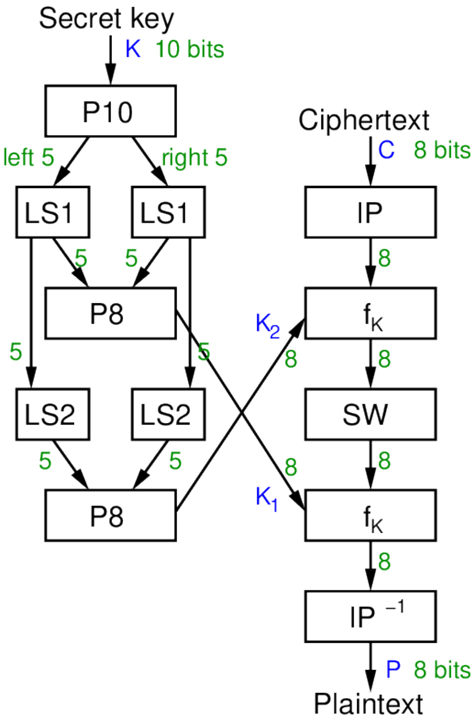


Figure 8.2: S-DES Key Generation and Decryption

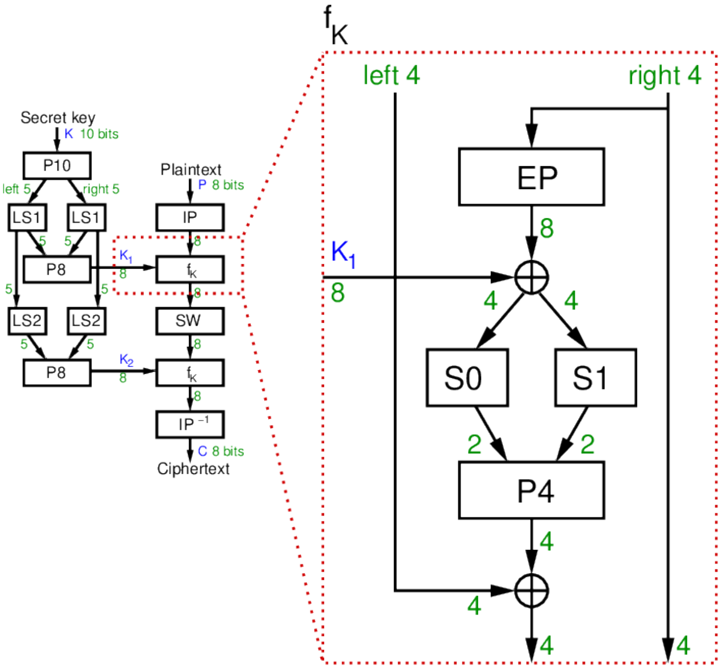


Figure 8.3: S-DES Round Function Details

Figure 8.3 shows the details of the round function, f_k. Note that the same steps are applied in the 2nd round, but instead K_2 is used as the round key. Operations include Expand and Permutate, XOR, S-boxes and a Permutation of 4 bits. The 8 bits output (left half and right half) are then input the the SWap block (swapping the two halves).

Definitions of the permutations and S-boxes follow.

Definition 8.1 (S-DES Permutations)**.** Permutations used in S-DES:

P10 (permutate)

```
Input :  1 2 3 4 5 6 7 8 9 10
Output:  3 5 2 7 4 10 1 9 8 6
```

P8 (select and permutate)

```
Input :  1 2 3 4 5 6 7 8 9 10
Output:  6 3 7 4 8 5 10 9
```

P4 (permutate)

```
Input :  1 2 3 4
Output:  2 4 3 1
```

EP (expand and permutate)

```
Input :  1 2 3 4
Output:  4 1 2 3 2 3 4 1
```

IP (initial permutation)

```
Input :  1 2 3 4 5 6 7 8
Output:  2 6 3 1 4 8 5 7
```

As an example, permutation P4 takes a 4-bit input and produces a 4-bit output. The 1st bit of the input becomes the 4th bit of the output. The 2nd bit of the input becomes the 1st bit of the output. The 3rd bit of the input becomes the 3rd bit of the output. The 4th bit of the input becomes the 1st bit on the output.

The permutations are fixed. That is they are always these exact permutations, and known by the encrypter, decrypter and attacker.

- LS-1: left shift by 1 position
- LS-2: left shift by 2 positions
- IP^{-1}: inverse of IP, such that $X = \mathrm{IP}^{-1}(\mathrm{IP}(X))$
- SW: swap the halves
- f_K: a round function using round key K
- F: internal function in each round

Definition 8.2 (S-DES S-Boxes)**.** S-Box considered as a matrix: input used to select row/column; selected element is output

4-bit input: $bit_1, bit_2, bit_3, bit_4$

$bit_1 bit_4$ specifies row (0, 1, 2 or 3 in decimal)

$bit_2 bit_3$ specifies column

$$S0 = \begin{bmatrix} 01 & 00 & 11 & 10 \\ 11 & 10 & 01 & 00 \\ 00 & 10 & 01 & 11 \\ 11 & 01 & 11 & 10 \end{bmatrix} \quad S1 = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 10 & 00 & 01 & 11 \\ 11 & 00 & 01 & 00 \\ 10 & 01 & 00 & 11 \end{bmatrix}$$

Exercise 8.1 (Encrypt with S-DES)**.** Show that when the plaintext `01110010` is encrypted using S-DES with key `1010000010` that the ciphertext obtained is `01110111`.

Video
Simplified DES Example (44 min; Jan 2016)
https://www.youtube.com/watch?v=3jGMCyOXOV8

Solution 8.1 (Encrypt with S-DES)**.** The input 10-bit key, K, is: `1010000010`. Then the steps for generating the two 8-bit round keys, K_1 and K_2, are:

1. Rearrange K using P10: `1000001100`
2. Left shift by 1 position both the left and right halves: `00001 11000`
3. Rearrange the halves with P8 to produce K_1: `10100100`
4. Left shift by 2 positions the left and right halves: `00100 00011`
5. Rearrange the halves with P8 to produce K_2: `01000011`

K_1 and K_2 are used as inputs in the encryption and decryption stages.
Now consider the 8-bit plaintext, P: `01110010`. Then the steps for encryption are:

1. Apply the initial permutation, IP, on P: `10101001`
2. Assume the input from step 1 is in two halves, L and R: `L=1010, R=1001`
3. Expand and permutate R using E/P: `11000011`
4. XOR input from step 3 with K_1: `10100100 XOR 11000011 = 01100111`
5. Input left halve of step 4 into S-Box S0 and right halve into S-Box S1:
   (a) For S0: 0110 as input: b_1, b_4 for row, b_2, b_3 for column
   (b) Row 00, column 11 → output is 10
   (c) For S1: 0111 as input:
   (d) Row 01, column 11 → output is 11
6. Rearrange outputs from step 5 (1011) using P4: `0111`
7. XOR output from step 6 with L from step 2: `0111 XOR 1010 = 1101`
8. Now we have the output of step 7 as the left half and the original R as the right half. Swap the halves and move to round 2: `1001 1101`
9. E/P with right half: `E/P(1101) = 11101011`
10. XOR output of step 9 with K_2: `11101011 XOR 01000011 = 10101000`
11. Input to S-Boxes:

   (a) For S0, 1010
   (b) Row 10, column 01 → output is 10
   (c) For S1, 1000
   (d) Row 10, column 00 → output is 11

12. Rearrange output from step 11 (1011) using P4: `0111`

13. XOR output of step 12 with left halve from step 8: `0111 XOR 1001 = 1110`

14. Input output from step 13 and right halve from step 8 into inverse IP
    (a) Input is: `1110 1101`
    (b) Output is: `01110111`

So our encrypted result of plaintext `01110010` with key `1010000010` is: `01110111`

In summary, S-DES:

- Educational encryption algorithm
- S-DES expressed as functions:

$$\text{ciphertext} = \text{IP}^{-1}(f_{K_2}(\text{SW}(f_{K_1}(\text{IP}(\text{plaintext})))))$$

$$\text{plaintext} = \text{IP}^{-1}(f_{K_1}(\text{SW}(f_{K_2}(\text{IP}(\text{ciphertext})))))$$

- Brute force attack on S-DES is easy since only 10-bit key
- If know plaintext and corresponding ciphertext, can we determine key? *Very hard*

The general design of S-DES follows the same principles as DES, although the algorithm parameters differ.

- S-DES vs DES
- Block size: 8 bits vs 64 bits
- Rounds: 2 vs 16
- IP: 8 bits vs 64 bits
- F: 4 bits vs 32 bits
- S-Boxes: 2 vs 8
- Round key: 8 bits vs 48 bits

The following section presents the details of DES. This is primarily for reference (or as evidence of the similarities and differences with S-DES). You are not expected to know the details of the DES operations.

8.3 Details of DES

DES was standardised by NIST as FIPS 46, with the latest version FIPS 46-3 withdrawn as a standard in 2005. The standard contains the technical details of DES, with the main figures repeated in the following. Most (if not all) NIST standards are in the public domain.

The following figures are directly from the FIPS 46-3 PDF and are Reprinted courtesy of the National Institute of Standards and Technology, U.S. Department of Commerce. Not copyrightable in the United States. For further explanation of DES, see the standard or various textbooks (such as Stallings).

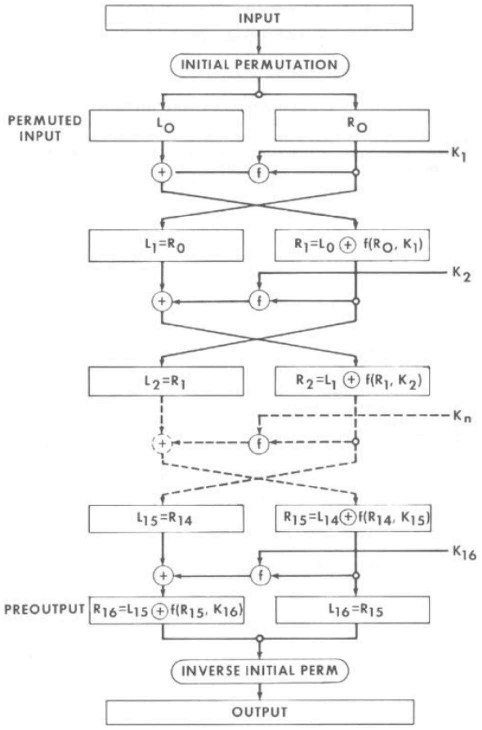


Figure 8.4: General DES Encryption Algorithm

Figure 8.4 shows the overall steps in DES encryption. The details of each block are shown in the following.

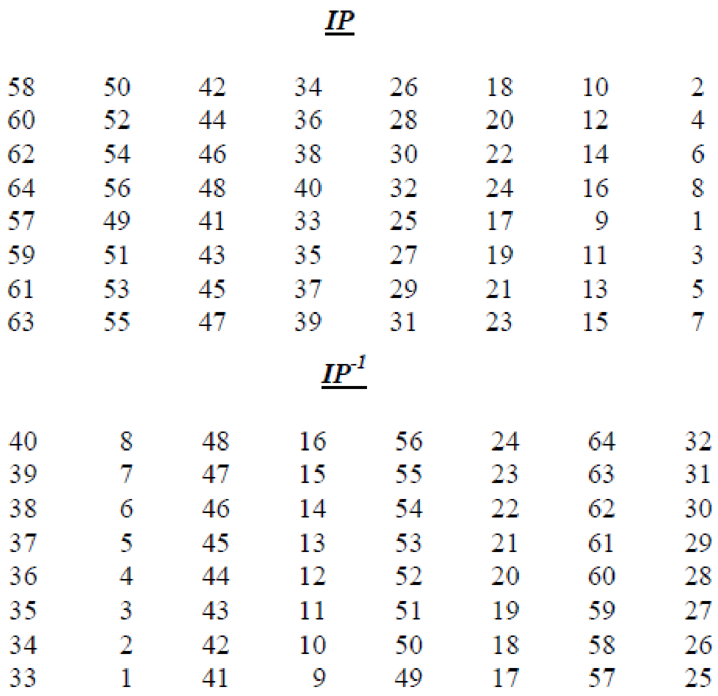
Figure 8.5 shows the initial permutation and it's inverse. The table is read row-by-row. So the 58th input bit becomes the 1st output bit. The 50th input bit becomes the 2nd output bit. And the 7th input bit becomes the 64th output bit.

Figure 8.6 shows the details of a single round of encruption, i.e. the round function. Similar to S-DES, it takes the right half, applies an expand and permutate (E), XOR with the round key, applies S-Boxes, and then a final permutate (P).

Figure 8.7 shows E and P which are used within a round of DES.

Figure 8.8 shows the first 4 S-Boxes. Each S-Box takes a 6 bit input. The first and last bit are used to determine the row, and the middle 4 bits determine the column. The result is a decimal values within the range 0 to 15, which determines the 4 bit output. See `https://en.wikipedia.org/wiki/DES_supplementary_material` for an example of reading the S-Boxes.

Figure 8.9 shows the last 4 S-Boxes.



IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Figure 8.5: Initial Permutation Tables for DES

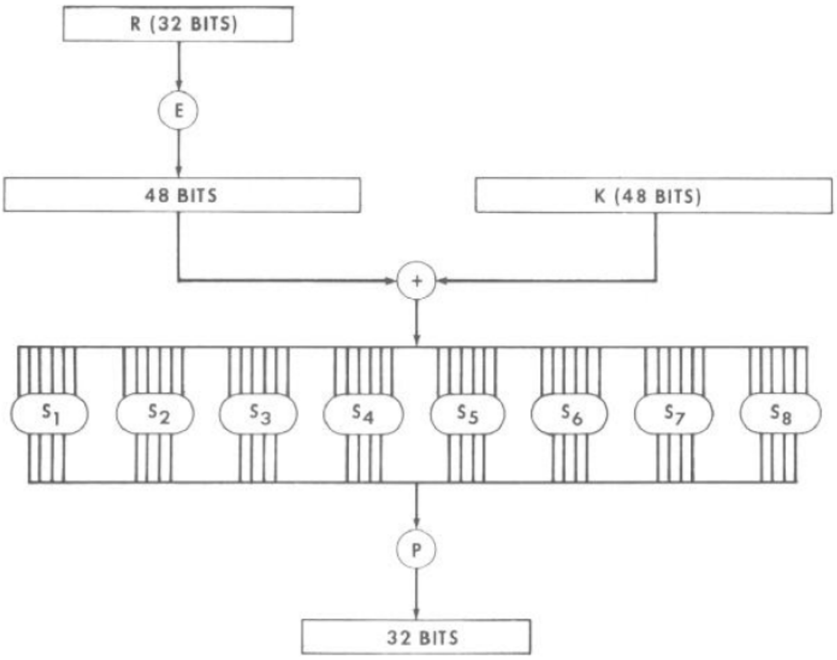
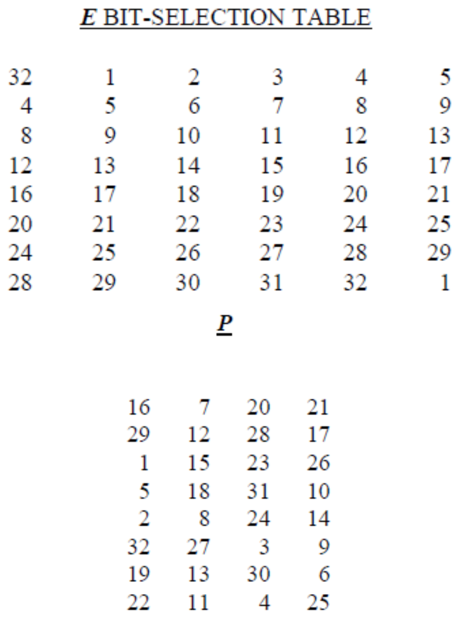


Figure 8.6: Calculation of F(R,K)



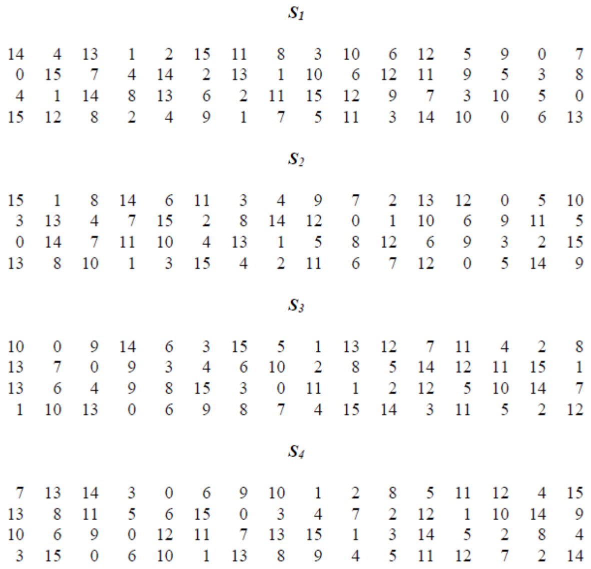
***E* BIT-SELECTION TABLE**

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Figure 8.7: Permutation Tables for DES



S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

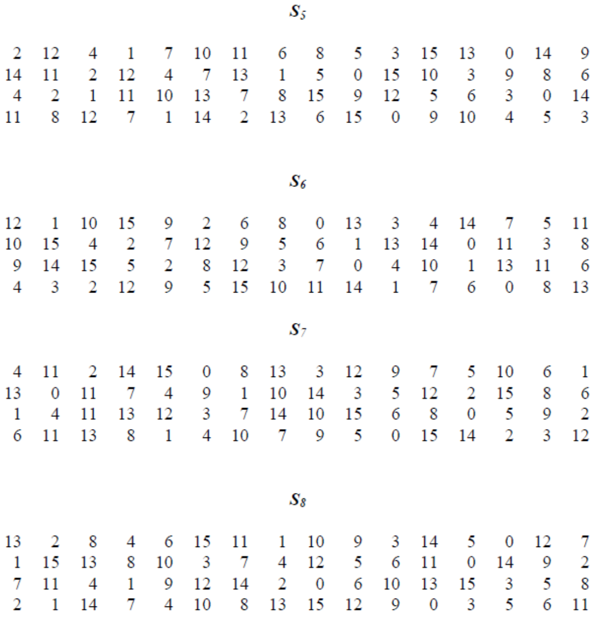
S_3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Figure 8.8: Definition of DES S-Boxes 1 to 4



S_5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

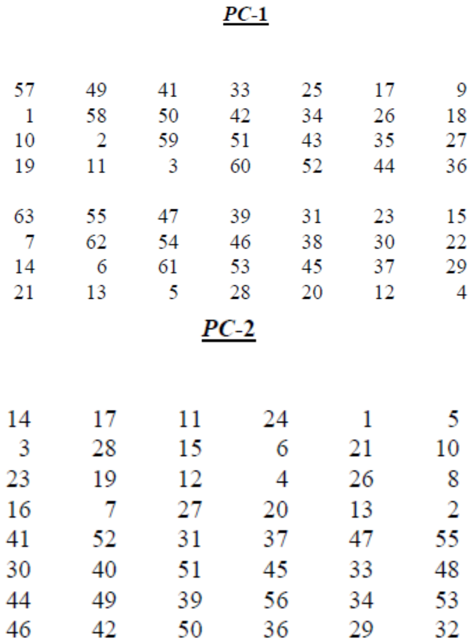
S_7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S_8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 8.9: Definition of DES S-Boxes 5 to 6



PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Figure 8.10: DES Permutated Choice 1 and 2

Figure 8.10 shows the Permutated Choices used in key generation.

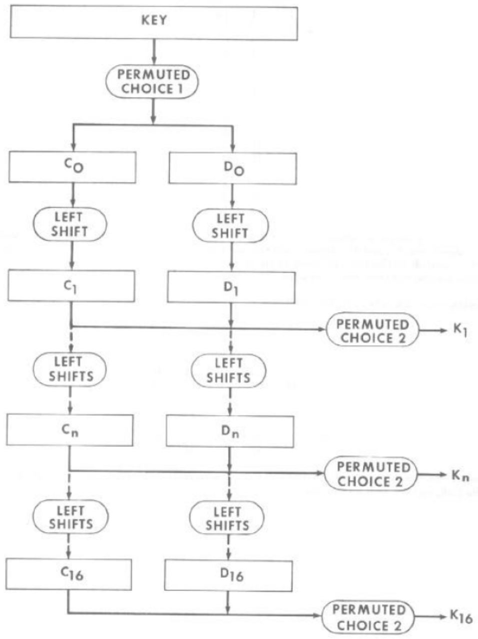
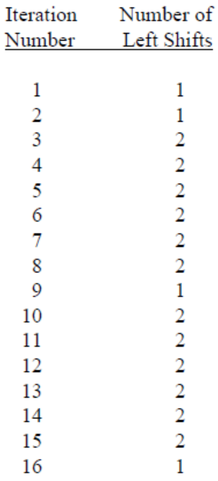


Figure 8.11: DES Key Generation Schedule

Figure 8.11 shows the overall key generation steps.



Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Figure 8.12: DES Schedule of Left Shifts in Key Generation

Figure 8.12 shows the schedule of left shifts indicating how many bits are shifted left when a Left Shift is applied in each round for key generation.

8.4 DES in OpenSSL

We will demonstrate several examples of using OpenSSL for DES encryption and decryption, including:

- Encrypt a file with a password using the `enc` operation
- Generate a random key using the `rand` operation
- Disable padding (with exact plaintext correct size)
- Encrypt with key and IV using `enc` operation
- View binary data (e.g. ciphertext) with `xxd`

8.4.1 DES Encryption Basics in OpenSSL

To demonstrate using OpenSSL to encrypt a file with DES, let's create an example plaintext message. You can use any file, but for the example, let's copy a plain text dictionary file most likely on your Linux system `/usr/share/dict/words`. We will name our plaintext file `plaintext1.in`. The file extension of `.in` is just to remember that this is the original plaintext input. After encrypting and decrypting, we may obtain outputs, for which we will use the extension `.out`. Remember, file extensions in Linux often do not matter (Chapter 4 of Network and Security in Linux explains basic Linux operations and files).

```
$ cp /usr/share/dict/words plaintext1.in
$ ls -l plaintext*
-rw-r--r-- 1 sgordon sgordon 938848 Jul 31 13:32 plaintext1.in
```

Now lets encrypt using DES. You can use the `list` command in OpenSSL to see the `-cipher-algorithms` and `-cipher-commands` (see Section 3.2.3). You will note there are different variants of DES, such as DES-ECB, DES-CBC and DES-CFB. The second identifier specifies the mode of operation. Modes of operation are covered in Chapter 11, but in short, these allow DES, which operates on 64-bit blocks, to be used to encrypt arbitrary sized plaintexts. The simplest mode of operation, but least secure, is Electronic Code Book (ECB). We will use ECB in this example.

Symmetric key encryption in OpenSSL is performed using the `enc` operation. In the simplest form, we specify the algorithm then the input file and output file (in our case, `ciphertext1.bin`). If we don't specify a secret key, then OpenSSL will prompt for a password and then convert that to a secret key.

```
$ openssl enc -des-ecb -in plaintext1.in -out ciphertext1.bin
enter des-ecb encryption password: password
Verifying - enter des-ecb encryption password: password
$ ls -l plaintext1.in ciphertext1.bin
-rw-rw-r-- 1 sgordon sgordon 938872 Jul 31 14:15 ciphertext1.bin
-rw-r--r-- 1 sgordon sgordon 938848 Jul 31 13:32 plaintext1.in
```

To decrypt, include the `-d` option:

```
$ openssl enc -d -des-ecb -in ciphertext1.bin -out plaintext1.out
enter des-ecb decryption password: password
$ ls -l plaintext1.in plaintext1.out
-rw-r--r-- 1 sgordon sgordon 938848 Jul 31 13:32 plaintext1.in
-rw-rw-r-- 1 sgordon sgordon 938848 Jul 31 14:18 plaintext1.out
$ diff plaintext1.in plaintext1.out
$ xxd -l 96 ciphertext1.bin
0000000: 5361 6c74 6564 5f5f f253 8361 b87d 1a3e Salted__.S.a.}.>
0000010: 30ed be95 5b38 ebf9 a013 ca64 bbf4 03ea 0...[8.....d....
0000020: 3ebb cdf8 483d 5a12 acd8 bc75 140c 920b >...H=Z....u....
0000030: da41 7376 edc3 b9bd 59c4 a5ce 0a67 408a .Asv....Y....g@.
0000040: d23e 10ee 7ac3 f5b6 4f09 4aaf 88e4 1f96 .>..z...O.J.....
0000050: 3171 7277 91a7 100c ac04 7871 dd39 cf4c 1qrw......xq.9.L
```

The lack of output from the `diff` command indicates the files `plaintext1.in` and `plaintext1.out` are identical. We've retrieved the original plaintext.

`xxd` was used to view the first 96 bytes, in hexadecimal, of the ciphertext. The first 8 bytes contain the special string Salted___ meaning the DES key was generated using a password and a salt. The salt is stored in the next 8 bytes of ciphertext, i.e. the value *f2538361b87d1a3e* in hexadecimal. So when decrypting, the user supplies the password and OpenSSL combines with the salt to determine the DES 64 bit key.

Section 8.4.2 shows a more detailed example where the key and IV are specified.

8.4.2 Symmetric Key Encryption Padding and Modes of Operation

Section 8.4.1 showed a simple method for performing symmetric key encryption with OpenSSL. Now we are going to consider some more details, in particular the role of padding and modes of operation.

Recall that block ciphers, like DES and AES, operate on fixed size blocks. For example, DES encrypts a 64 bit (or 8 Byte) block of plaintext. But commonly the plaintext we want to encrypt is larger than a single block. *Modes of operation*, such as ECB, Cipher Block Chaining (CBC) and Counter mode (CTR), are used to apply the block cipher across multiple blocks. That is, encrypt the first 8 Bytes of plaintext with DES, then encrypt the next 8 Bytes of plaintext (or related data) with DES, and combine them together according to some algorithm. The details of modes of operation are covered in Chapter 11.

A related issue is that often the full plaintext will not be an integer multiple of blocks. For example, a 50 Byte file consists of 6 by 8 Byte blocks with 2 Bytes in the 7th block. Padding is needed to fill out that 7th block. By default, OpenSSL performs padding for you. However if you are sure you have a correct length plaintext (integer multiple of blocks), you can omit padding. This is useful to perform simple exploration of the output.

The following shows an example of using OpenSSL without padding, and demonstrates the weakness of the ECB mode of operation.

To get started, we need a plaintext message to encrypt. The first command below generates a message (saving to a file), and the subsequent commands show us some information about the message/file.

```
$ echo -n "Hello. This is our super secret message. Keep it secret please.
    Goodbye." > plaintext.txt
$ cat plaintext.txt
Hello. This is our super secret message. Keep it secret please. Goodbye.
$ wc -m plaintext.txt
72 plaintext.txt
```

```
$ ls -l
total 4
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:39 plaintext.txt
$ xxd -c 8 plaintext.txt
0000000: 4865 6c6c 6f2e 2054 Hello. T
0000008: 6869 7320 6973 206f his is o
0000010: 7572 2073 7570 6572 ur super
0000018: 2073 6563 7265 7420 secret
0000020: 6d65 7373 6167 652e message.
0000028: 204b 6565 7020 6974 Keep it
0000030: 2073 6563 7265 7420 secret
0000038: 706c 6561 7365 2e20 please.
0000040: 476f 6f64 6279 652e Goodbye.
$ xxd -b -c 8 plaintext.txt
0000000: 01001000 01100101 01101100 01101100 01101111 00101110 00100000
    01010100 Hello. T
0000008: 01101000 01101001 01110011 00100000 01101001 01110011 00100000
    01101111 his is o
0000010: 01110101 01110010 00100000 01110011 01110101 01110000 01100101
    01110010 ur super
0000018: 00100000 01110011 01100101 01100011 01110010 01100101 01110100
    00100000 secret
0000020: 01101101 01100101 01110011 01110011 01100001 01100111 01100101
    00101110 message.
0000028: 00100000 01001011 01100101 01100101 01110000 00100000 01101001
    01110100 Keep it
0000030: 00100000 01110011 01100101 01100011 01110010 01100101 01110100
    00100000 secret
0000038: 01110000 01101100 01100101 01100001 01110011 01100101 00101110
    00100000 please.
0000040: 01000111 01101111 01101111 01100100 01100010 01111001 01100101
    00101110 Goodbye.
```

The meaning of the preceding output is:

1. Create a short text message with `echo`. The `-n` option is used to ensure no newline is added to the end. There are two things about this message that will be important later: the length is a multiple of 8 characters (9 by 8 characters) and the word `secret` appears twice (in particular positions).

2. Display the message on the screen with `cat`.

3. Count the number of characters with `wc`.

4. View the file size with `ls`.

5. Show the message in hexadecimal and binary using `xxd`. From now on, we'll only look at the hexadecimal values (not binary).

To encrypt with DES-ECB we need a secret key (as well as IV). You can choose your own values. For security, they should be randomly chosen. We saw in Chapter 3 different ways to generate random values. Let's use OpenSSL's `rand` twice: the first will be for the secret key and the second for the IV.

```
$ openssl rand -hex 8
001e53e887ee55f1
$ openssl rand -hex 8
a499056833bb3ac1
```

Now encrypt the plaintext using DES-ECB. The IV and Key are taken from the outputs OpenSSL PRNG above. Importantly, we use the `-nopad` option at the end:

```
$ openssl enc -des-ecb -e -in plaintext.txt -out ciphertext.bin -iv
    a499056833bb3ac1 -K 001e53e887ee55f1 -nopad
```

Now look at the output ciphertext. First note it is the same length as the plaintext (as expected, when no padding is used). And on initial view, the ciphertext looks random (as expected). But closer inspection you see there is some structure: the 4th and 7th lines of the xxd output are the same. This is because it corresponds to the encryption of the same original plaintext " secure " (recall that word was repeated in the plaintext, in the positions such that it is in a 64-bit block). Since ECB is used, repetitions in input plaintext blocks will result in repetitions in output ciphertext blocks. This is insecure (especially for long plaintext). Another mode of operation, like CBC, should be used.

```
$ ls -l
total 8
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:42 ciphertext.bin
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:39 plaintext.txt
$ xxd -c 8 ciphertext.bin
0000000: 56dc b368 d9ef 0793 V..h....
0000008: 7be4 a87d e26d c2f1 {..}.m..
0000010: e042 bbe6 9e00 6d37 .B....m7
0000018: f1e9 7163 cb4a 38d8 ..qc.J8.
0000020: 5394 a92f 8cf2 ac72 S../...r
0000028: 5064 be07 f67c d807 Pd...|..
0000030: f1e9 7163 cb4a 38d8 ..qc.J8.
0000038: a31c 0efd cd0b dd03 ........
0000040: 0486 7e2d 00ad 762d ..~-..v-
```

Now lets decrypt:

```
$ openssl enc -des-ecb -d -in ciphertext.bin -out received.txt -iv
    a499056833bb3ac1 -K 001e53e887ee55f1 -nopad
```

And look at the decrypted value. Of course, it matches the original plaintext message.

```
$ ls -l
total 12
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:42 ciphertext.bin
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:39 plaintext.txt
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:43 received.txt
$ cat received.txt
Hello. This is our super secret message. Keep it secret please. Goodbye.
$ xxd -c 8 received.txt
0000000: 4865 6c6c 6f2e 2054 Hello. T
0000008: 6869 7320 6973 206f his is o
0000010: 7572 2073 7570 6572 ur super
0000018: 2073 6563 7265 7420 secret
```

```
0000020: 6d65 7373 6167 652e message.
0000028: 204b 6565 7020 6974  Keep it
0000030: 2073 6563 7265 7420  secret
0000038: 706c 6561 7365 2e20 please.
0000040: 476f 6f64 6279 652e Goodbye.
```

Now lets try and decrypt again, but this time using the wrong key. I've changed the last hexadecimal digit of the key from "1" to "2".

```
$ openssl enc -des-ecb -d -in ciphertext.bin -out received2.txt -iv
    a499056833bb3ac1 -K 001e53e887ee55f2 -nopad
```

Looking at the decrypted message, it is random. We didn't obtain the original plaintext. Normally, when padding is used, OpenSSL adds a checksum when encrypting which allows, after decrypting, incorrect deciphered messages to be automatically detected.

```
$ ls -l
total 16
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:42 ciphertext.bin
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:39 plaintext.txt
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:46 received2.txt
-rw-r--r-- 1 sgordon sgordon 72 Nov 11 16:43 received.txt
$ xxd -c 8 received2.txt
0000000: 0346 e59e c22d 403f .F...-@?
0000008: 63ff 28fd eb6b 387d c.(..k8}
0000010: b52f d595 06c0 342f ./....4/
0000018: f419 3569 e383 c857 ..5i...W
0000020: 0a77 0b49 6f62 cb64 .w.Iob.d
0000028: 8265 d419 51f3 ea12 .e..Q...
0000030: f419 3569 e383 c857 ..5i...W
0000038: f296 33f3 5cf4 d359 ..3.\..Y
0000040: e205 4018 0ce0 34f5 ..@...4.
```

However the checksum used within OpenSSL is not perfect, so it shouldn't be relied upon for secure authentication (i.e. checking the received message is correct). Chapter 17 discusses different ways for the receiver to be sure they have obtained the original message.

Video
DES Encryption using OpenSSL (13 min; Jan 2012)
https://www.youtube.com/watch?v=VdE21ku7SMs

8.4.3 DES OpenSSL Exercises

Exercise 8.2 (DES Key Generation)**.** Generate a shared secret key to be used with DES and share it with another person.

Solution 8.2 (DES Key Generation)**.** It is important that any symmetric key is generated randomly. Using OpenSSL `rand` operation is a good approach. See Section 3.2.4 and/or Section 8.4.2 for examples.

The key must be 64 bits (8 bytes or 16 hex digits).

Exercise 8.3 (DES Encryption)**.** Create a message in a plain text file and after using DES, send the ciphertext to the person you shared the key with.

Solution 8.3 (DES Encryption)**.** See OpenSSL examples in Section 8.4.2. The sender and receiver should agree upon the mode of operation, an IV (recommended to be random in general, although not needed for ECB) and the use of padding (recommended to be used).

Exercise 8.4 (DES Decryption)**.** Decrypt the ciphertext you received.

Solution 8.4 (DES Decryption)**.** See OpenSSL examples in Section 8.4.2.

8.5 DES in Python

The Python Cryptography library includes symmetric key encryption using various algorithms. DES is covered under TripleDES. That is, using TripleDES with a 64 bit key is equivalent to using DES. See the examples for generic symmetric encryption at:

- cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/

Chapter 9

Advanced Encryption Standard

This chapter provides details of Advanced Encryption Standard (AES), with concepts demonstrated via a simplified, educational version called Simplified Advanced Encryption Standard (S-AES). Many of the details serve mainly as reference, with little discussion.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

9.1 Overview of AES

As the 56-bit key of DES became a practical limitation, researchers and standard bodies worked towards new block ciphers.

- 1977: DES (56-bit key). NIST published.
- 1991: IDEA, similar to DES, secure but patent issues
- 1999: 3DES (168-bit key). NIST recommended 3DES be used (DES only for legacy systems)
  - 3DES was considered secure (apart from special case attacks)
  - But 3DES is very slow, especially in software
  - DES and 3DES use 64-bit blocks – larger block sizes required for efficiency
- 1997: NIST called for proposals for new Advanced Encryption Standards
  - Proposals made public and evaluations performed
- 2001: Selected *Rijndael* as the algorithm for AES

The process for determining the algorithm to be selected for AES was performed as a multi-round competition run by NIST. There are varying criteria for selecting a winner.

- Original NIST criteria:
  - Security: effort to cryptoanalyse algorithm, randomness, ...

File: crypto/aes.tex, r1980

  - Cost: royalty-free license, computationally efficient, ...
  - Algorithm and implementation characteristics: flexibility (different keys/blocks, implement on different systems), simplicity, ...

- 21 candidate algorithms reduced to 5
- Updated NIST evaluation criteria for 5 algorithms:
  - General Security
  - Software and hardware implementations (needs to be efficient)
  - Low RAM/ROM requirements (e.g. for smart cards)
  - Ability to change keys quickly
  - Potential to use parallel processors

Rijndael, a proposal by Vincent Rijmen and Joan Daemen, was selected the winner for the following reasons:

- Security: good, no known attacks
- Software implementation: fast, can make use of parallel processors
- Hardware implementation: fastest of all candidates
- Low memory requirements: good, except encryption and decryption require separate space
- Timing and Power analysis attacks: easiest to defend against
- Key flexibility: supports on-the-fly change of keys and different size of keys/blocks

Key parameters of Rijndael were the block and key sizes. While the Rijndael support various sizes, the eventual NIST standard settling on a single block size with three key lengths.

- NIST Advanced Encryption Standard, FIPS-197, 2001
- Three variations of same algorithm standardised
  - AES-128: 128-bit key, 10 rounds
  - AES-192: 192-bit key, 12 rounds
  - AES-256: 256-bit key, 14 rounds
- AES uses 128-bit block size for all variations
- Simplified Advanced Encryption Standard (S-AES) used to understand AES (educational only)
- For details of AES see the Stallings textbook, AES on Wikipedia or the AES standard from NIST

9.2 Simplified-AES

- Educational purposes only. Mohammad A. Musa , Edward F. Schaefer and Stephen Wedig (2003) A Simplified AES Algorithm and its Linear and Differential Cryptanalyses, Cryptologia, 27:2, 148-177, DOI: 10.1080/0161-110391891838
- Input: 16-bit block of plaintext; 16-bit key
- Output: 16-bit block of ciphertext
- Operations:
  - Add Key: XOR of a 16-bit key and 16-bit state matrix
  - Nibble Substitution: S-Box table lookup that swaps nibbles (4 bits)
  - Shift Row: shift of nibbles in a row
  - Mix Column: re-order columns
  - Rotate Nibbles: swap the nibbles
- 3 rounds (although they don't contain same operations)

S-AES operates on 16-bit blocks, with some operations on 8-bit words and others on 4-bit nibbles. For example, a 16-bit block is equivalent to two 8-bit words or four 4-bit nibbles.

Figure 9.1, Figure 9.2, and Figure 9.3 show the encryption, decryption and key generation algorithms, respectively. The operations used in the algorithms are defined later.

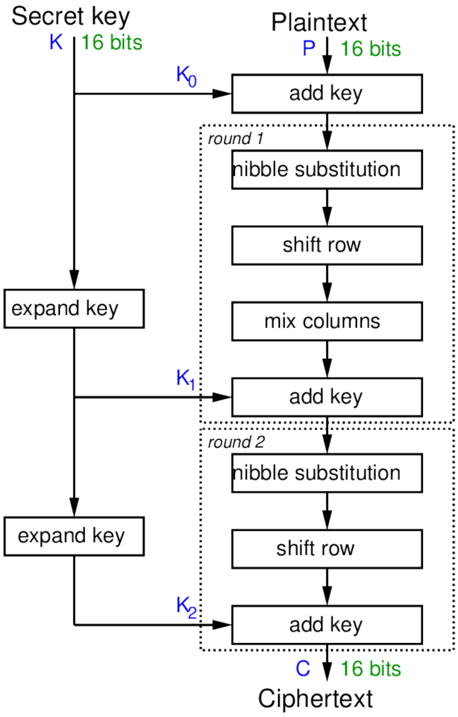


Figure 9.1: S-AES Encryption

Figure 9.1 shows the overall steps for S-AES and key expansion and encryption. The key generation takes a 16-bit secret key and expands that into 3 16-bit round keys. The first round key K_0 is simple the original key. The next two round keys, K_1 and K_2 are generated by an expansion algorithm. Figure 9.3 shows that algorithm for K_1.

S-AES encryption operates on 16-bit blocks of plaintext. To encrypt, there is an initial *add key*, and then two rounds, where the 2nd round does not include the *mix columns* operation.

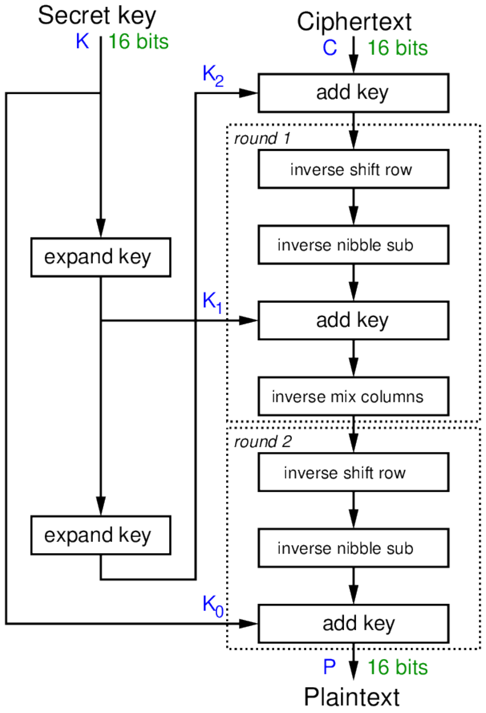


Figure 9.2: S-AES Decryption

Figure 9.2 shows the decryption operations. Note that it is similar to encryption in reverse, with all operations replaced with their inverse operations. The same round keys are used as in encryption, but in the opposite order.

Figure 9.3 shows the key generation operations for generated round key K_1. Similar steps are used to generate K_2, where the input is K_1 and a different round constant.

Definition 9.1 (S-AES State Matrix)**.** S-AES operates on a 16-bit state matrix, viewed as 4 nibbles

$$\begin{bmatrix} b_0b_1b_2b_3 & b_8b_9b_{10}b_{11} \\ b_4b_5b_6b_7 & b_{12}b_{13}b_{14}b_{15} \end{bmatrix} = \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix}$$

While S-AES operates on 16-bits at a time, those bits are viewed as a state matrix of 4 nibbles. Note the matrix is filled columnwise, with the first 8 bits (2 nibbles) in the first column.

The following shows operations based on the state matrix.

Definition 9.2 (S-AES Shift Row, Add Key and Rotate Nibbile operations)**.** S-AES Shift Row:

$$\begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} \rightarrow \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,1} & S_{1,0} \end{bmatrix}$$

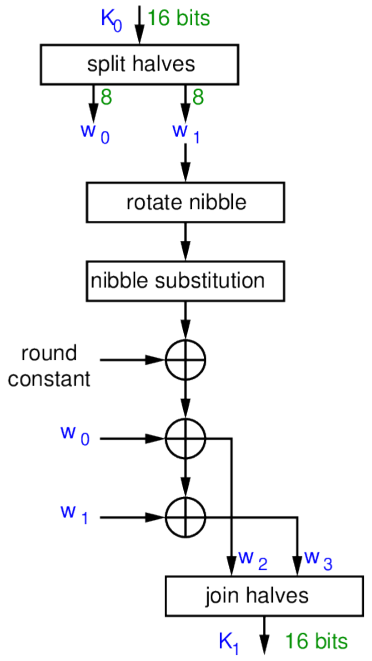


Figure 9.3: S-AES Key Generation for Round 1

S-AES Add Key: Exclusive OR (XOR)
S-AES Rotate Nibble: swap the two nibbles
S-AES Nibble Substitution: apply S-Box on each nibble
S-AES Round Constant 1: `10000000`
S-AES Round Constant 2: `00110000`

Shift Row swaps the 2nd nibble with the 4th nibble. Add Key is a bitwise XOR. The round constants are used in the key generation.

Definition 9.3 (S-AES S-Boxes)**.** S-Box considered as a matrix: input used to select row/column; selected element is output
Input: 4-bit nibble, $bit_1, bit_2, bit_3, bit_4$
$bit_1 bit_2$ specifies row
$bit_3 bit_4$ specifies column

$$\text{encrypt}: \begin{bmatrix} 1001 & 0100 & 1010 & 1011 \\ 1101 & 0001 & 1000 & 0101 \\ 0110 & 0010 & 0000 & 0011 \\ 1100 & 1110 & 1111 & 0111 \end{bmatrix}$$

$$\text{decrypt}: \begin{bmatrix} 1010 & 0101 & 1001 & 1011 \\ 0001 & 0111 & 1000 & 1111 \\ 0110 & 0000 & 0010 & 0011 \\ 1100 & 0100 & 1101 & 1110 \end{bmatrix}$$

The left-most 2 bits in a nibble determine the row, and the right-most 2 bits in the nibble determine the column. The output nibble is based on the S-Box. The Inverse S-Box is used in decryption.

Definition 9.4 (S-AES Mix Columns)**.** Mix the columns in the state matrix be performing a matrix multiplication.

Mix Columns:

$$\begin{bmatrix} S'_{0,0} & S'_{0,1} \\ S'_{1,0} & S'_{1,1} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix}$$

Inverse Mix Columns:

$$\begin{bmatrix} S'_{0,0} & S'_{0,1} \\ S'_{1,0} & S'_{1,1} \end{bmatrix} = \begin{bmatrix} 9 & 2 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix}$$

Galois Field GF(2^4) is used for addition and multiplication operations.

S' denotes the output from the mixing of columns, e.g. $S'_{0,0} = (1 \times S_{0,0}) + (4 \times S_{1,0})$. Importantly, the resulting addition and multiplication operations are in Galois Field GF(2^4). We do not cover (Galois) fields, however in Number Theory we saw modular arithmetic with mod n where all operations produced results within 0 to n. This is a simple case of a field, i.e. all operations produce answers within some finite range. GF(2^4) means all answers will be within range 0 to 15.

GF(2^4) addition is equivalent to bitwise XOR. However GF(2^4) multiplication is more complicated. Therefore, for the purpose of demonstrating S-AES, a simplified view of the mix column operations with a table lookup for multiplication is shown in the following.

Definition 9.5 (S-AES Mix Columns (Simple))**.** Mix the columns in the state matrix be performing the following calculations.

Mix Columns:

$$S'_{0,0} = S_{0,0} \oplus (0100 \times S_{1,0})$$
$$S'_{1,0} = (0100 \times S_{0,0}) \oplus S_{1,0}$$
$$S'_{0,1} = S_{0,1} \oplus (0100 \times S_{1,1})$$
$$S'_{1,1} = (0100 \times S_{0,1}) \oplus S_{1,1}$$

Inverse Mix Columns:

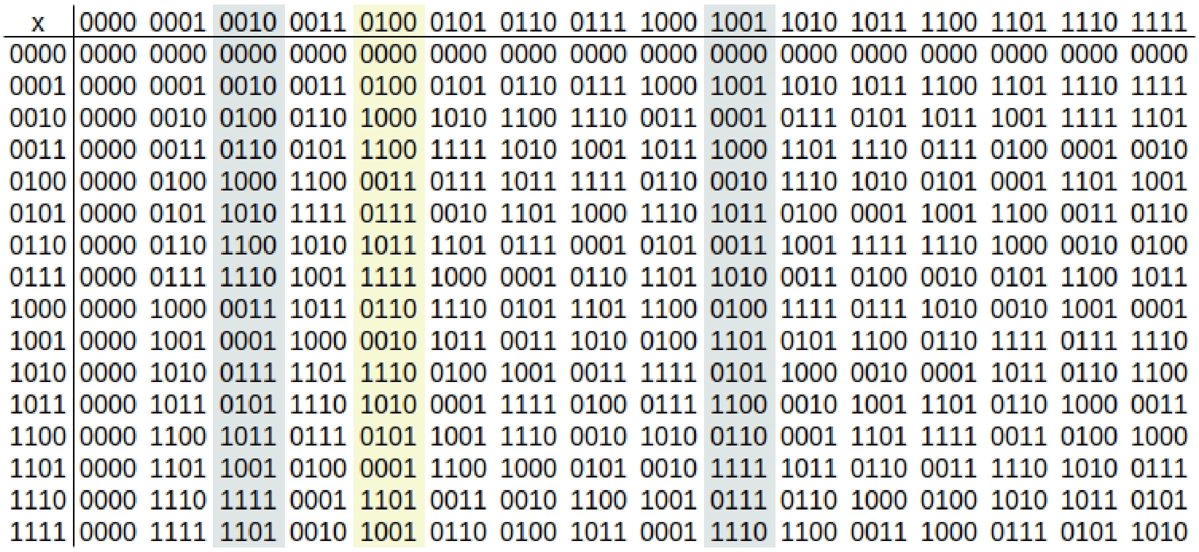
$$S'_{0,0} = (1001 \times S_{0,0}) \oplus (0010 \times S_{1,0})$$
$$S'_{1,0} = (0010 \times S_{0,0}) \oplus (1001 \times S_{1,0})$$
$$S'_{0,1} = (1001 \times S_{0,1}) \oplus (0010 \times S_{1,1})$$
$$S'_{1,1} = (0010 \times S_{0,1}) \oplus (1001 \times S_{1,1})$$

For multiplication, lookup using Figure 9.4.

Figure 9.4 shows the GF(2^4) multiplication table in binary. The green column is used in encryption (Mix Columns) and the two blue columns are used in decryption (Inverse Mix Columns). For example with encryption, when multiplying a value by 4 (0100 in binary), lookup the value in the first column (e.g. 0111) and the answer will be in the green column (e.g. 1111).

Now let's compare S-AES to the real AES, specifically AES-128.

- S-AES



x	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0001	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0010	0000	0010	0100	0110	1000	1010	1100	1110	0011	0001	0111	0101	1011	1001	1111	1101
0011	0000	0011	0110	0101	1100	1111	1010	1001	1011	1000	1101	1110	0111	0100	0001	0010
0100	0000	0100	1000	1100	0011	0111	1011	1111	0110	0010	1110	1010	0101	0001	1101	1001
0101	0000	0101	1010	1111	0111	0010	1101	1000	1110	1011	0100	0001	1001	1100	0011	0110
0110	0000	0110	1100	1010	1011	1101	0111	0001	0101	0011	1001	1111	1110	1000	0010	0100
0111	0000	0111	1110	1001	1111	1000	0001	0110	1101	1010	0011	0100	0010	0101	1100	1011
1000	0000	1000	0011	1011	0110	1110	0101	1101	1100	0100	1111	0111	1010	0010	1001	0001
1001	0000	1001	0001	1000	0010	1011	0011	1010	0100	1101	0101	1100	0110	1111	0111	1110
1010	0000	1010	0111	1101	1110	0100	1001	0011	1111	0101	1000	0010	0001	1011	0110	1100
1011	0000	1011	0101	1110	1010	0001	1111	0100	0111	1100	0010	1001	1101	0110	1000	0011
1100	0000	1100	1011	0111	0101	1001	1110	0010	1010	0110	0001	1101	1111	0011	0100	1000
1101	0000	1101	1001	0100	0001	1100	1000	0101	0010	1111	1011	0110	0011	1110	1010	0111
1110	0000	1110	1111	0001	1101	0011	0010	1100	1001	0111	0110	1000	0100	1010	1011	0101
1111	0000	1111	1101	0010	1001	0110	0100	1011	0001	1110	1100	0011	1000	0111	0101	1010

Figure 9.4: GF(2^4) Multiplication Table used in S-AES

  - 16-bit key, 16-bit plaintext/ciphertext
  - 2 rounds: first with all 4 operations, last with 3 operations
  - Round key size: 16 bits
  - Mix Columns: arithmetic over GF(2^4)

- AES-128
  - 128-bit key, 128-bit plaintext/ciphertext
  - 10 rounds: first 9 with all 4 operations, last with 3 operations
  - Round key size: 128 bits
  - Mix Columns: arithmetic over GF(2^8)

- Principles of operation are the same

9.3 Simplified-AES Example

Exercise 9.1 (Encrypt with S-AES)**.** Show that when the plaintext `1101 0111 0010 1000` is encrypted using Simplified-AES with key `0100 1010 1111 0101` that the ciphertext obtained is `0010 0100 1110 1100`.

Solution 9.1 (Encrypt with S-AES)**.** See the PDF of the solution at:
https://sandilands.info/sgordon/teaching/reports/simplified-aes-example-v2.pdf

9.4 AES in OpenSSL

9.4.1 AES Encryption Basics in OpenSSL

You can use AES in a similar manner to DES in OpenSSL (see Section 8.4), that is, using the `enc` operation. The following is an example of AES encryption with a 128 bit key and CTR mode of operation. In addition to the key, an IV is needed. The `plaintext2.in` file is just an example, in fact it is obtained from copying the actual OpenSSL binary, e.g. `cp /usr/bin/openssl plaintext2.in`.

```
$ openssl enc -aes-128-ctr -in plaintext2.in -out ciphertext2.bin -K
    0123456789abcdef0123456789abcdef -iv 00000000000000000000000000000000
$ openssl enc -d -aes-128-ctr -in ciphertext2.bin -out plaintext2.out -K
    0123456789abcdef0123456789abcdef -iv 00000000000000000000000000000000
$ ls -l *2*
-rw-rw-r-- 1 sgordon sgordon 513208 Jul 31 14:29 ciphertext2.bin
-rwxr-xr-x 1 sgordon sgordon 513208 Jul 31 13:32 plaintext2.in
-rw-rw-r-- 1 sgordon sgordon 513208 Jul 31 14:30 plaintext2.out
$ diff plaintext2.in plaintext2.out
$ xxd -l 96 ciphertext2.bin
0000000: 06ee 8984 3a69 ac84 d388 ce61 110a 6274 ....:i.....a..bt
0000010: c1ed f9ed f193 f2d2 bf8d 29e2 1577 5d32 ..........)..w]2
0000020: 1e25 cc36 bb37 baa7 eb65 402b a8ef 421b .%.6.7...e@+..B.
0000030: a6f7 073c a08a e698 747d 5153 8df1 ed88 ...<....t}QS....
0000040: 1131 f4e0 2014 1392 ee36 2b54 27eb ca72 .1.. ....6+T'..r
0000050:␣4b88␣e623␣ed28␣2da7␣87cd␣0c1a␣5441␣5d7c␣␣K..#.(-.....TA]|
```

Both the Key (note uppercase `-K`) and IV were specified on the command line as a hexadecimal string. With AES-128, they must be 32 hex digits (128 bits). You may choose any value you wish.

Use the `list` operation in OpenSSL to see the variants of AES supported by OpenSSL (see Section 3.2.3).

9.4.2 AES Performance Benchmarking

OpenSSL has a built-in operation for performance testing. It encrypts random data over short period, measuring how many bytes can be encrypted per second. It can be used to compare the performance of different algorithms, and compare the performance of different computers.

To run performance tests across a large set of algorithms, simple use the `speed` operation. Note that it may take a few minutes:

```
$ openssl speed
...
```

You can select the algorithms to test, e.g. AES, DES and Message Digest 5 hash function (MD5):

```
$ openssl speed aes-128-cbc des md5
...
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
```

```
md5            68101.86k  199387.83k  444829.62k  639419.85k  734323.76k
des cbc        76810.00k   78472.53k   78442.77k   79241.85k   78440.45k
des ede3       28883.98k   29585.17k   29640.69k   29499.08k   29740.52k
aes-128 cbc   138894.09k  150561.30k  154512.15k  155203.81k  155590.46k
```

The output shows the progress, the versions and options used for OpenSSL and then a summary table at the end. Focus on the summary table, and the last line (for aes-128-cbc) in the example above. The speed test encrypts as many b Byte input plaintexts as possible in a period of 3 seconds. Different size inputs are used, i.e. $b = 16$, 64, 256, 1024 and 8192 Bytes. The summary table reports the encryption speed in Bytes per second. So if 25955833 16-Byte plaintext values are encrypted in 3 seconds, then the speed reported in the summary table is $25955833 \times 16 \div 3 \approx 138$ million Bytes per second. You can see that value (138,894.09kB/s) in the table above. So AES using 128 bit key and CBC can encrypt about 138 MB/sec when small plaintext values are used and 155 MB/sec when plaintext values are 8192 Bytes.

Normally OpenSSL implements all algorithms in software. However recent Intel CPUs include instructions specifically for AES encryption, a feature referred to as AES-NI. If an application such as OpenSSL uses this special instruction, then part of the AES encryption is performed directly by the CPU. This is usually must faster (compared to using general instructions). To run a speed test that uses the Intel AES-NI, use the `evp` option:

```
$ openssl speed -evp aes-128-cbc
...
type             16 bytes     64 bytes    256 bytes  1024 bytes  8192 bytes
aes-128-cbc     689927.75k  729841.81k  745383.38k  747226.84k  747784.87k
```

Compare the values to the original results. In the original test we achieved 138 MB/sec. Using the Intel AES hardware encryption we get a speed of 689 MB/sec, about 5 times faster.

9.4.3 AES OpenSSL Exercises

Exercise 9.2 (AES Key Generation)**.** Generate a shared secret key to be used with AES and share it with another person.

Solution 9.2 (AES Key Generation)**.** It is important that any symmetric key is generated randomly. Using OpenSSL `rand` operation is a good approach. See Section 3.2.4 for examples.

The users need to select a key length: 128, 192 or 256 bits.

Exercise 9.3 (AES Encryption)**.** Create a message in a plain text file and after using AES, send the ciphertext to the person you shared the key with.

Solution 9.3 (AES Encryption)**.** See OpenSSL examples in Section 9.4.1. The sender and receiver should agree upon the mode of operation, an IV (recommended to be random in general, although not needed for ECB) and the use of padding (recommended to be used).

Exercise 9.4 (AES Decryption)**.** Decrypt the ciphertext you received.

Solution 9.4 (AES Decryption)**.** See OpenSSL examples in Section 9.4.1.

Exercise 9.5 (AES Performance Benchmarking)**.** Perform speed tests on AES using both the software and hardware implementations (if available). Compare and discuss the impact of the following on performance: key length; software vs hardware; different computers (e.g. compare the performance with another person).

Solution 9.5 (AES Performance Benchmarking)**.** See OpenSSL examples in Section 9.4.2. The performance of AES-128, AES-192 and AES-256 should be compared. Also, compare software implementation of AES (default when running OpenSSL) with the hardware implementation (`-evp`) if supported by your computer.

9.5 AES in Python

The Python Cryptography library includes symmetric key encryption using various algorithms, including AES. See the examples for generic symmetric encryption at:

- `https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/`

Chapter 10

Pseudorandom Number Generators

To be added in the future.

File: crypto/prng.tex, r1769

Chapter 11

Block Cipher Modes of Operation

This chapter presents common modes of operation available with symmetric block ciphers. Modes of operation allow the block ciphers to be applied to inputs greater than the block size. The difference in designs lead to different security and performance tradeoffs. This chapter is primarily for reference, presenting the modes but with little explanation of each.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

11.1 Block Ciphers with Multiple Blocks

Block ciphers operate on fixed length inputs, so the question arises of how are they used to encrypt arbitrary length inputs?

- Block cipher: operates on fixed length b-bit input to produce b-bit ciphertext
- What about encrypting plaintext longer than b bits?
- Naive approach: Break plaintext into b-bit blocks (padding if necessary) and apply cipher on each block independently
  - ECB
- Security issues arise:
  - Repetitions of input plaintext blocks produces repetitions of output ciphertext blocks
  - Repetitions (patterns) in ciphertext are bad!
- Different modes of operation have been developed
- Tradeoffs between security, performance, error handling and additional features (e.g. include authentication)

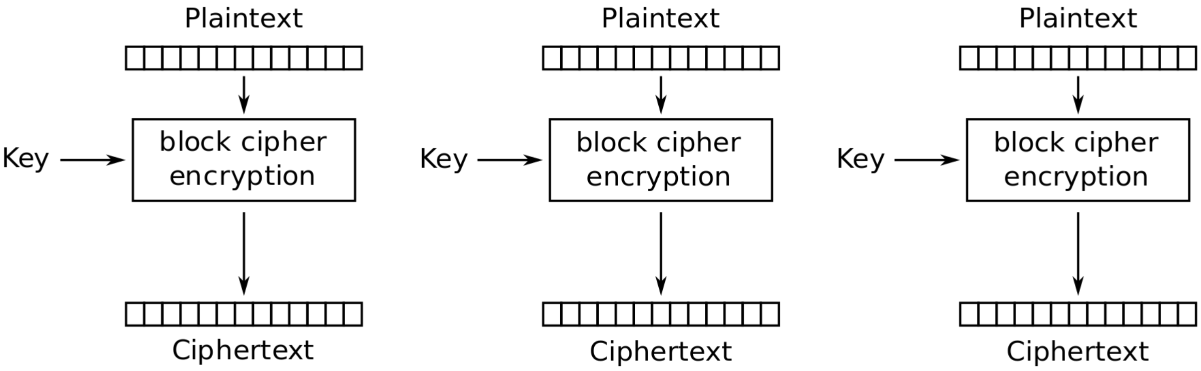
File: crypto/modes.tex, r1949

We will not cover each mode of operation in detail, but rather present them so you are aware of some of the common modes. For more technical details of some of these modes of operation, including discussion of padding, error propagation and the use of initialisation vectors, see NIST Special Publication 800-38A Recommendations for Block Cipher Modes of Operation: Methods and Techniques. Additional (newer) modes of operation are in the NIST SP 800-38 series, such as 800-38C CCM, 800-38D GCM and 800-38E XTS-AES.

11.2 Electronic Code Book

Figure 11.1 and Figure 11.2 show the Electronic Code Book (ECB) mode of operation applied for encryption and decryption, respectively.

- Each block of 64 plaintext bits is encoded independently using same key
- Typical applications: secure transmission of single values (e.g. encryption key)
- Problem: with long message, repetition in plaintext may cause repetition in ciphertext



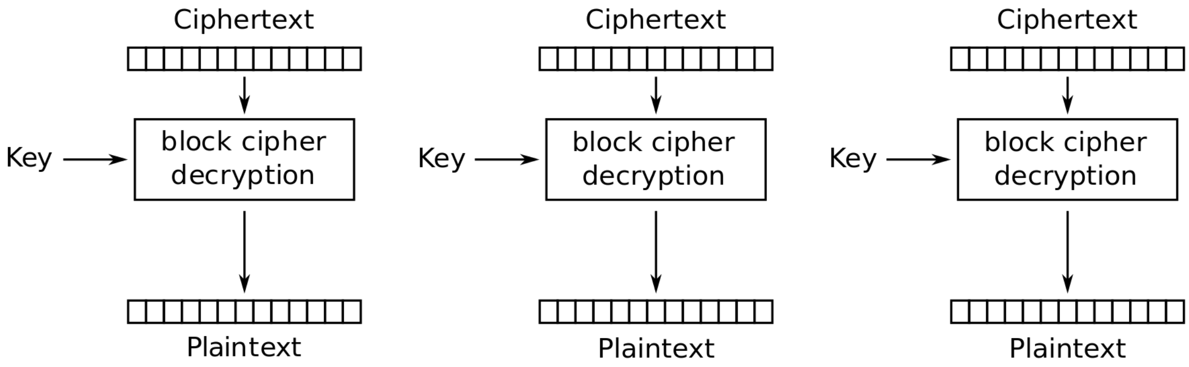
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:ECB_encryption.svg, public domain

Figure 11.1: ECB Encryption

11.3 Cipher Block Chaining Mode

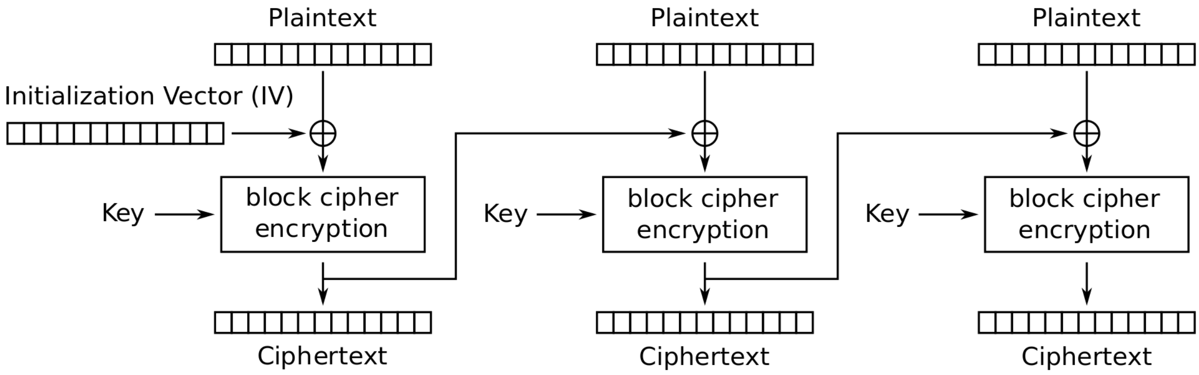
Figure 11.3 and Figure 11.4 show the Cipher Block Chaining (CBC) mode of operation applied for encryption and decryption, respectively.

- Input to encryption algorithm is XOR of next 64-bits plaintext and preceding 64-bits ciphertext
- Typical applications: General-purpose block-oriented transmission; authentication
- Initialisation Vector (IV) must be known by sender/receiver, but secret from attacker



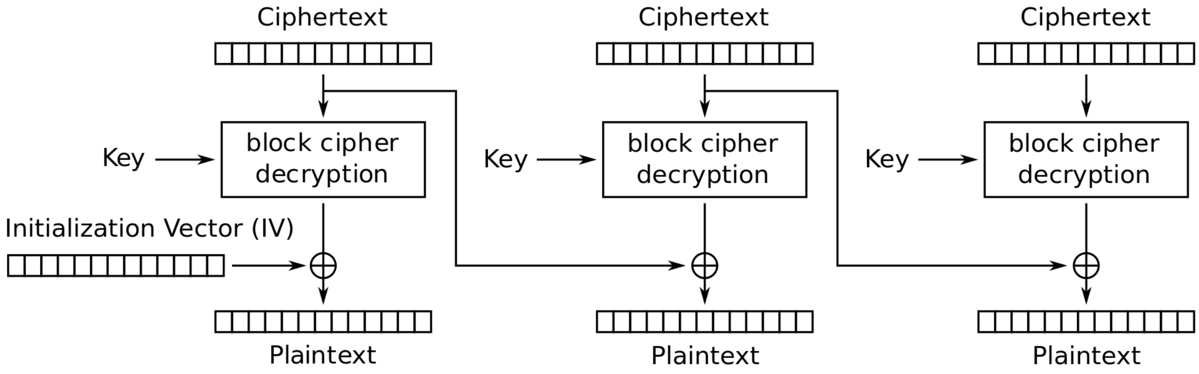
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:ECB_decryption.svg, public domain

Figure 11.2: ECB Decryption



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CBC_encryption.svg, public domain

Figure 11.3: CBC Encryption



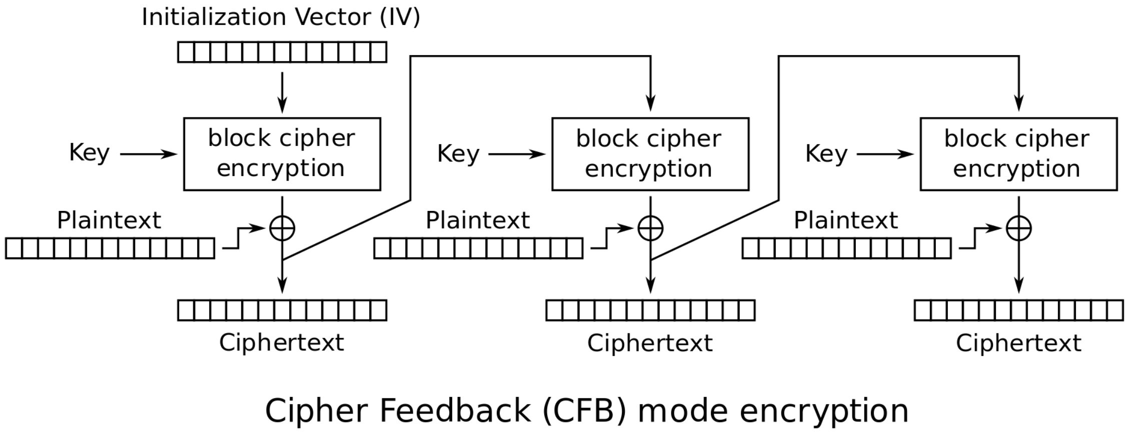
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CBC_decryption.svg, public domain

Figure 11.4: CBC Decryption

11.4 Cipher Feedback Mode

Figure 11.5 and Figure 11.6 show the Cipher Feedback mode (CFB) mode of operation applied for encryption and decryption, respectively.

- Converts block cipher into stream cipher
  - No need to pad message to integral number of blocks
  - Operate in real-time: each character encrypted and transmitted immediately
- Input processed s bits at a time
- Preceding ciphertext used as input to cipher to produce pseudo-random output
- XOR output with plaintext to produce ciphertext
- Typical applications: General-purpose stream-oriented transmission; authentication



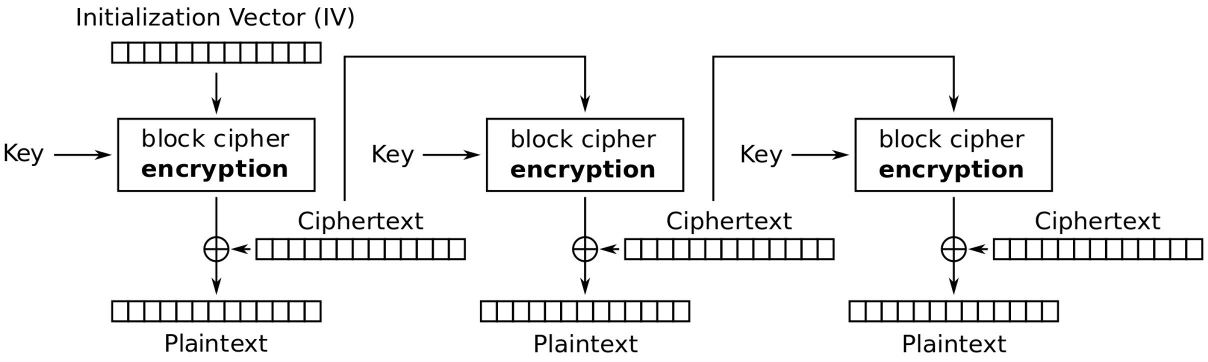
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CFB_encryption.svg, public domain

Figure 11.5: CFB Encryption

11.5 Output Feedback Mode

Figure 11.7 and Figure 11.8 show the Output Feedback mode (OFB) mode of operation applied for encryption and decryption, respectively.

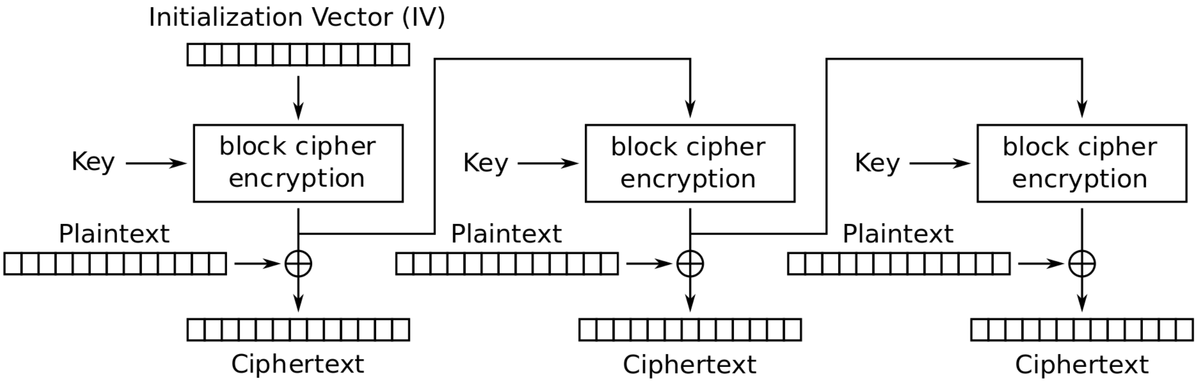
- Converts block cipher into stream cipher
- Similar to CFB, except input to encryption algorithm is preceding encryption output
- Typical applications: stream-oriented transmission over noisy channels (e.g. satellite communications)



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CFB_decryption.svg, public domain

Figure 11.6: CFB Decryption

- Advantage compared to OFB: bit errors do not propagate
- Disadvantage: more vulnerable to message stream modification attack



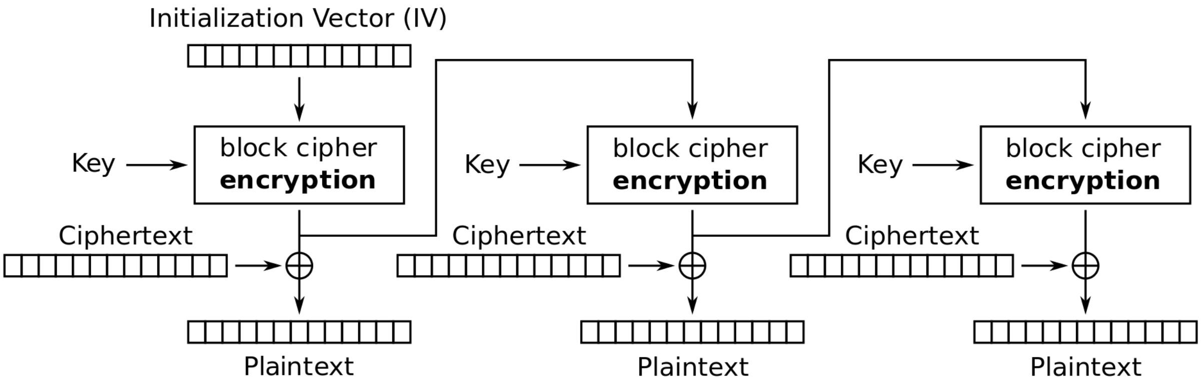
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:OFB_encryption.svg, public domain

Figure 11.7: OFB Encryption

11.6 Counter Mode

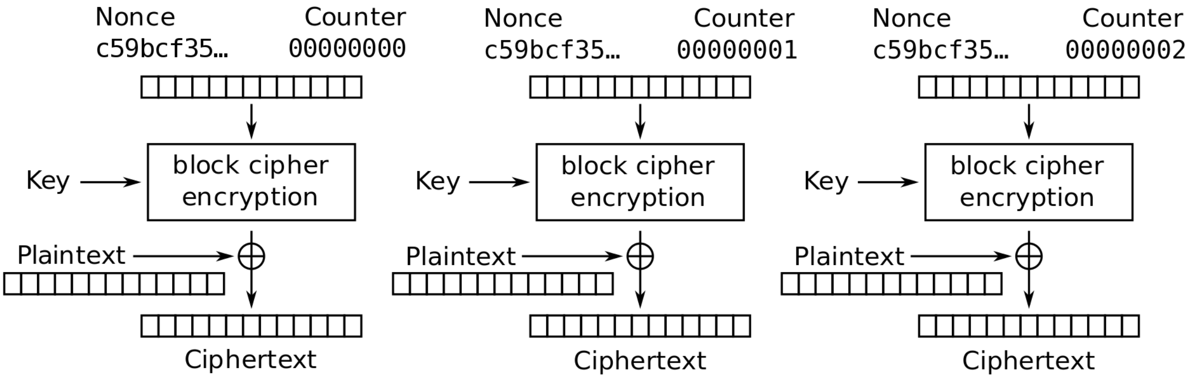
Figure 11.9 and Figure 11.10 show the Counter mode (CTR) mode of operation applied for encryption and decryption, respectively.

- Converts block cipher into stream cipher
- Each block of plaintext XORed with encrypted counter
- Typical applications: General-purpose block-oriented transmission; useful for high speed requirements
- Efficient hardware and software implementations
- Simple and secure



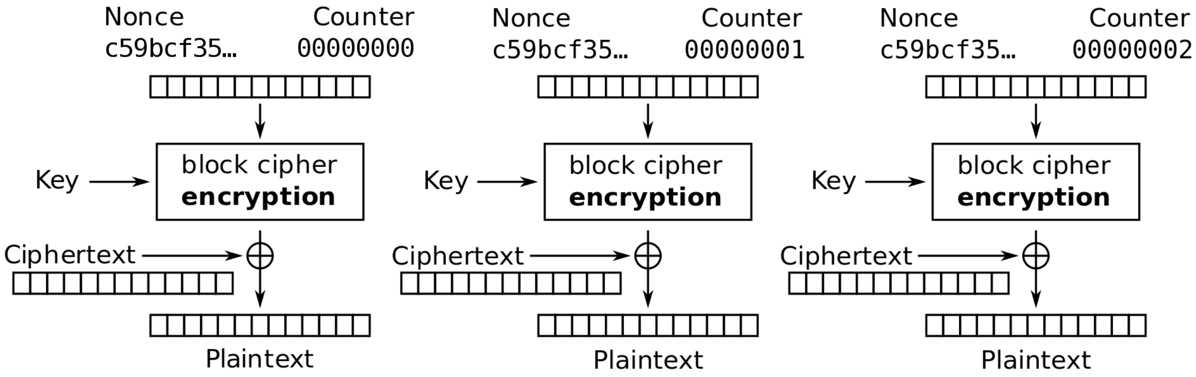
Credit: Wikimedia https://commons.wikimedia.org/wiki/File:OFB_decryption.svg, public domain

Figure 11.8: OFB Decryption



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CTR_encryption_2.svg, public domain

Figure 11.9: CTR Encryption



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CTR_decryption_2.svg, public domain

Figure 11.10: CTR Decryption

11.7 XTS-AES

XTS-AES is a mode of operation designed for AES to be used to encrypt stored data (e.g. disk drives). Compared to CBC, it improves the ability for a receiver to detect if the ciphertext has been changed.

- XTS-AES designed for encrypting stored data (as opposed to transmitted data)
- Overcomes potential attack on CBC whereby one block of the ciphertext is changed by the attacker, and that change does not affect all other blocks
- See Stallings Chapter 6.7 for details and differences to transmitted data encryption

Part IV

Public Key Cryptography

Chapter 12

Public Key Cryptography

This chapter summarises key concepts in public key cryptography. These concepts will be demonstrated when looking at specific algorithms, including RSA (Chapter 13), Diffie-Hellman Key Exchange (Chapter 14) and Elliptic Curve Cryptography (Chapter 15).

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

12.1 Concepts of Public Key Cryptography

We have seen how symmetric key cryptography can be used for encryption. Now let's look at an alternative approach, public key cryptography.

- Symmetric Key Encryption
  - Same key used for encryption and decryption
  - Key is randomly generated (e.g. by sender)
  - Problem: How does receiver securely obtain secret key?
- Public (or asymmetric) key encryption
  - Two different, but mathematically related keys
  - One key (public) for encryption, another key (private) for decryption
  - Since encrypt key is public, key exchange is not a problem
  - Ciphers designed around math problems
  - Problem: Performance: much, much slower than symmetric

With symmetric key encryption, assume the sender generates a random key. The receiver of the encrypted data must also know that key in order to decrypt the data. But how does the receiver learn the key? If the sender sends the key *unencrypted* then an attacker can learn the key and it is no longer secret. If the sender encrypts the key, then the same problem arises: how do they get the second key (which is used to encrypt the first key) to the receiver?

Public key encryption can solve this problem, as we will see in the following slides.

File: crypto/public.tex, r1944

Symmetric key encryption has been the main form of cryptography for a long time. It wasn't until the 1960's and 1970's that public key cryptography was designed.

- Every user has their own key pair: (PU, PR)
  - Keys are generated using known algorithm (they are not chosen randomly like symmetric keys)
- Public key, PU
  - Available to everyone, e.g. in email signature, on website, in newspaper
- Private key, PR
  - Secret, known only by owner, e.g. access restricted file on computer
- Ciphers: if encrypt with one key in the pair, can only successfully decrypt with the other key in the pair

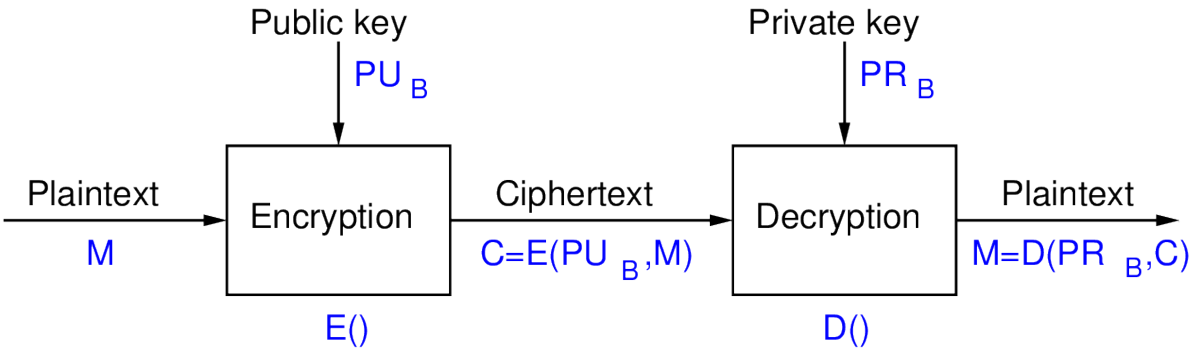
Consider all the students in the class. With public key crypto, each student would generate their own key pair. They could tell everyone their public key (e.g. yell it out in class, print on the screen and show), but they must keep their private key secret. Note that the keys are related: an algorithm is used to generate them (they are not randomly chosen like symmetric key encryption secret keys). That algorithm must be designed such that it is practically impossible for someone to find the private key if they know the public key.

The encryption/decryption algorithms in public key crypto are designed such that if you encrypt plaintext with one key in the pair, then you can only successfully decrypt the ciphertext if using the other key from that pair. For example, if you encrypt a message with the public key of Steve, then you can only decrypt the ciphertext if you know the private key of Steve.

Some public key ciphers also work in the other direction: if you encrypt a message with the private key of Steve, then you can only decrypt the ciphertext if you know the public key of Steve. We will see this in digital signatures.

This assumes User A (on the left) already knows the public key of user B. Since it is PUBLIC there is no problem with A knowing B's public key. However in practice, there are problems with A being sure that the public key does indeed belong to B (maybe it is someone pretending to be B). We don't cover that here, but in the chapter on digital certificates we will see this issue (of knowing who's public key it is) be addressed.

- Public key ciphers consist of:
  - Key generation algorithm
  - Encryption algorithm
  - Decryption algorithm
- Designed around computationally hard mathematical problems
- Very hard to solve without key, i.e. trapdoor functions
  - Finding prime factors of large integers



- User A is sender, user B is receiver
- Encrypt using receivers public key, PU_B
- Decrypt using receivers private key, PR_B
- Only B has PR_B, therefore only B can successfully decrypt → confidentiality

Figure 12.1: Confidentiality with Public Key Crypto

  - Solving logarithms in modulo arithmetic
  - Solving logarithms on elliptic curves

The details of the algorithms are covered in subsequent chapters.

- RSA (Rivest Shamir Adleman)
  - Security depends on difficult to factor large integers
  - Widely used for digital signatures
- Diffie-Hellman
  - Security depends on difficult to solve logarithms in modulo arithmetic
  - Widely used for secret key exchange
- Elliptic Curve
  - Security depends on difficulty to solve logarithms on elliptic curve
  - Newer, used in signatures and key exchange
  - Smaller keys is benefit

Video
Concepts of Public Key Cryptography (21 min; Apr 2021)
https://www.youtube.com/watch?v=9ZFm9i_uYvM

Chapter 13

RSA

This chapter presents the RSA algorithm, as an example of public key cryptography.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

13.1 RSA Algorithm

The Rivest Shamir Adleman cipher (RSA) is the most widely known public key cryptosystem.

- Created Ron Rivest, Adi Shamir and Len Adleman in 1978
- Formed RSA Security (company) in 1982 to commercialise products
- Most widely used public-key algorithm
- RSA is a block cipher: plaintext and ciphertext are integers

As we will see, the plaintext and ciphertext are integers. Any data can be represented in binary, and then split into blocks, where each block is taken as an input to RSA.

More information about Rivest, Shamir and Adleman is given in Chapter C.

- Step 1: Users generated RSA key pairs using RSA Key Generation Algorithm
- Step 2: Users exchange public key
- Step 3: Sender encrypts plaintext using RSA Encryption Algorithm
- Step 4: Receiver decrypts ciphertext using RSA Decryption Algorithm

The following will show the algorithms used in steps 1, 3 and 4. For now we assume the users can exchange public keys, noting that public keys do not need to be kept secret. For example, one method to exchange public keys over a network is to simply email the public key, unencrypted. It doesn't matter if an attacker intercepts the public key, since, by definition, it is public to everyone.

Later we will see that the exchange of public keys is in fact harder than it seems.

File: crypto/rsa.tex, r1945

Video
Introduction to RSA (4 min; Apr 2021)
https://www.youtube.com/watch?v=29Jpc-rvH8w

Algorithm 13.1 (RSA Key Generation)**.** Each user generates their own key pair

1. Choose primes p and q
2. Calculate $n = pq$
3. Select e: $gcd(\phi(n), e) = 1, 1 < e < \phi(n)$
4. Find $d \equiv e^{-1} \pmod{\phi(n)}$

The user keeps p, q and d private. The values of e and n can be made public.

- Public key of user, $PU = \{e, n\}$
- Private key of user $PR = \{d, n\}$

Note that the private key includes both d and n, however the same n is also included in the public key. So while n is included in the private key, it is not actually private. This describes the conceptual view of the RSA public and private key. Implementations of RSA may store additional information in the keys, especially the private key.

Video
RSA Key Generation Algorithm (6 min; Apr 2021)
https://www.youtube.com/watch?v=bKuaAv8LsFY

Exercise 13.1 (RSA Key Generation)**.** Assume user A chose the primes $p = 17$ and $q = 11$. Find the public and private keys of user A.

Solution 13.1 (RSA Key Generation)**.** First calculate n:

$$\begin{aligned} n &= p \times q \\ &= 17 \times 11 \\ &= 187 \end{aligned}$$

Now find $\phi(n)$ using the property of Euler's totient (Definition 5.8):

$$\begin{aligned} \phi(p \times q) &= \phi(p) \times \phi(q) \\ &= (p-1) \times (q-1) \\ &= (17-1) \times (11-1) \\ &= 16 \times 10 \\ &= 160 \end{aligned}$$

Next we choose a number relatively prime with 160, which will be e. Or in other words, the greatest common divisor of e and 160 is 1. There are multiple values possible for e. We need to choose just one value, and at this point, any of those values. Let's start small. As 160 is even, the even numbers will not be relatively prime with 160 so we can ignore them. What about 3? As 3 is prime and is not a divisor of 160, then 3 and 160 are relatively prime. So $e = 3$ is a valid choice. There are other valid choices (e.g. 7, 9, 11, ...), but we will go with 3.

Now we need to find the multiplicative inverse of 3 in mod 160. That is, find a d such that:

$$3 \times d \pmod{160} \equiv 1$$

The extended Euclidean algorithm can efficiently find a multiplicative inverse. But for now, as we are using small numbers, we can use trial and error. Note that the condition can be satisfied if we can find a d that satisfies the following, for an integer a:

$$3 \times d = (a \times 160) + 1$$

Therefore we can try integers a, and check if $(a \times 160) + 1$ is divisible by 3.

$$(1 \times 160) + 1 = 161, \text{ which is not divisible by } 3$$

$$(2 \times 160) + 1 = 321, \text{ which is divisible by } 3, \text{ giving } 107$$

Therefore $d = 107$.

We now have the RSA key pair of user A:

$$PU_A = \{e = 3, n = 187\} \text{ and } PR_A = \{d = 107, n = 187\}$$

Video
RSA Key Generation Example (14 min; Feb 2015)
https://www.youtube.com/watch?v=_57utzfyyPY

Algorithm 13.2 (RSA Encryption and Decryption)**.** Encryption of plaintext M, where $M < n$:

$$C = M^e \bmod n$$

Decryption of ciphertext C:

$$M = C^d \bmod n$$

Note the conceptual simplicity of the encryption and decryption algorithms, compared to DES and AES. Also note that the decryption algorithm is in fact identical to encryption—it is only the variable names that have changed.

Video
RSA Encryption and Decryption (2 min; Apr 2021)
https://www.youtube.com/watch?v=lQLJy6XVRuY

For a RSA to be usable it must meet the following usability and security requirements:

1. Successful decryption: Possible to find values of e, d, n such that $M^{ed} \bmod n = M$ for all $M < n$

2. Successful decryption: Encryption with one key of a key pair (e.g. PU) can only be successfully decrypted with the other key of the key pair (e.g. PR)

3. Computational efficiency: Easy to calculate $M^e \bmod n$ and $C^d \bmod n$ for all values of $M < n$

4. Secure: Infeasible to determine d or M from known information e, n and C

5. Secure: Infeasible to determine d or M given known plaintext, e.g. (M_1, C_1)

We will not show how RSA meets these requirements yet (it is covered in more depth later), but RSA does indeed meet these requirements.

The 1st requirement is that if a message is encrypted, then the decryption of the resulting ciphertext will produce the original message.

The 2nd requirement is that you can only use keys in the same key pair; using the wrong key will produce incorrect results.

The 3rd requirement is that users can easily perform the encrypt and decrypt operations. By "easily" we mean within reasonable time (i.e. seconds, not thousands of years).

The 4th requirement is that an attacker cannot find the private value d or the message.

The 5th requirement is that, even if the attacker knows old plaintext values and the corresponding ciphertext (which was obtained using the same key pair), they should not be able to find d or M.

Looking at the algorithms it is not immediately obvious how the security requirements are met. That is because, for example, the encryption algorithm is an equation with 4 variables (C, M, e, n), of which 3 are known to the attacker. Why can't the attacker re-arrange the equation and find the value of the unknown variable C? We will see some analysis of the security later.

- RSA encryption uses one key of a key pair, while decryption must use the other key of that same key pair

- RSA works no matter the order of the keys

- RSA for confidentiality of messages

  - Encrypt using the public key of receiver
  - Decrypt using the private key of receiver

- RSA for authentication of messages

  - Encrypt using the private key of the sender (called signing)
  - Decrypt using the public key of the sender (called verification)

- In practice, RSA is primarily used for authentication, i.e. sign and verifying messages

Why does confidentiality work? Since the receiver is the only user that knows their private key, then they are the only user that can decrypt the ciphertext.

Why does authentication work? Since the sender is the only user that knows their private key, then they are the only user that can sign the message/plaintext. And the receiver can verify it came from that user if the signature decrypts successful with the sender's public key.

Figures 13.1 and 13.2 illustrate how the key pair is used in RSA to provide either confidentiality or authentication. Note that such a feature (ability to use keys in either direction) is including in some, but not all, public key cryptography ciphers.

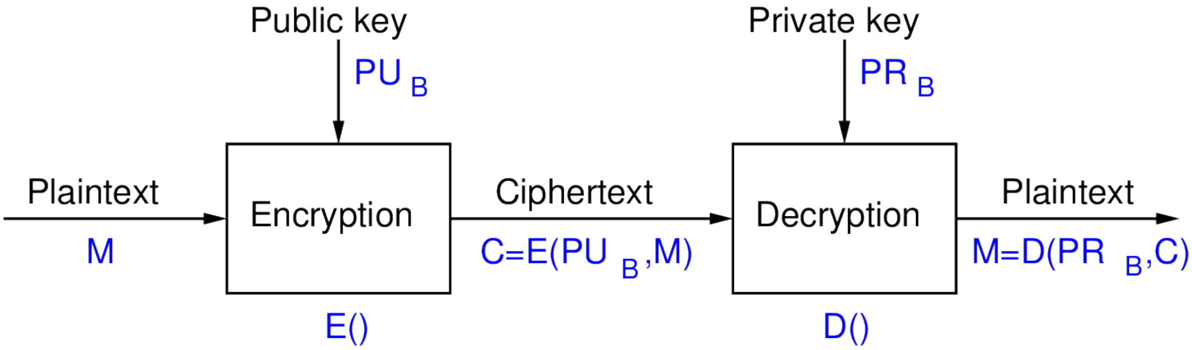


Figure 13.1: RSA used for Confidentiality

Figure 13.1 shows RSA used to provide confidentiality of the message M. User A is on the left and user B is on the right. The operations E() and D() correspond to the encrypt and decrypt algorithms of RSA, respectively. User A encrypts the message using user B's public key, PU_B. The ciphertext is sent to user B. User B then decrypts using their own private key, PR_B.

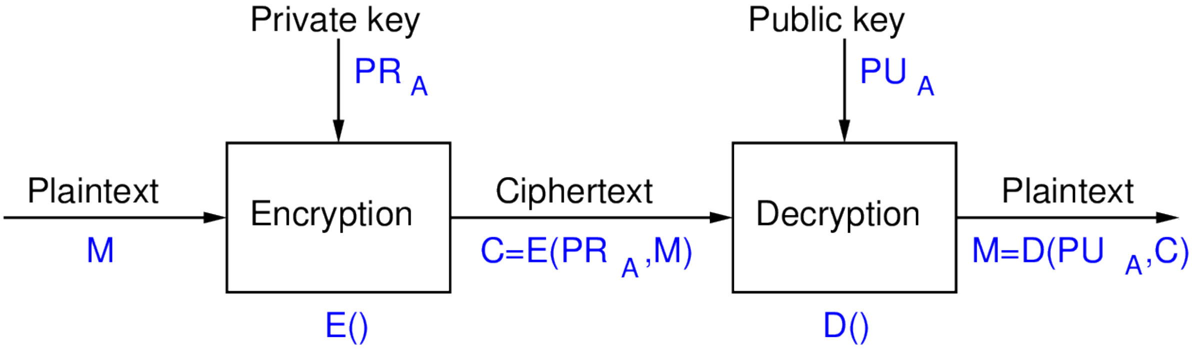


Figure 13.2: RSA used for Authentication

Figure 13.2 shows RSA used to provide authentication of the message M. The operations E() and D() correspond to the encrypt and decrypt algorithms of RSA, respectively, however they are more commonly referred to as signing and verification operations, respectively. User A encrypts/signs the message using their own private key, PR_A. The ciphertext/signed message is sent to user B. User B then decrypts/verifies using user A's public key, PU_A.

Exercise 13.2 (RSA Encryption for Confidentiality)**.** Assume user B wants to send a confidential message to user A, where that message, M is 8. Find the ciphertext that B will send A.

Solution 13.2 (RSA Encryption for Confidentiality)**.** For confidentiality, the sender encrypts using the receiver's public key. From the previous key generation exercise, the public key of user A is $PU_A = \{e = 3, n = 187\}$. With $M = 8$, the RSA encryption algorithm can be applied:

$$\begin{aligned} C &= M^e \bmod n \\ &= 8^3 \bmod 187 \\ &= 512 \bmod 187 \\ &= 138 \end{aligned}$$

Therefore the ciphertext is $C = 138$.

Video
RSA Encryption Example (11min; Feb 2015)
https://www.youtube.com/watch?v=fdGGErmf9E8

Exercise 13.3 (RSA Decryption for Confidentiality)**.** Show that user A successfully decrypts the ciphertext.

Solution 13.3 (RSA Decryption for Confidentiality)**.** User A receives the ciphertext, $C = 138$ from B, and decrypts using their own private key $PR_A = \{d = 107, n = 187\}$.

$$\begin{aligned} M &= C^d \bmod n \\ &= 138^{107} \bmod 187 \end{aligned}$$

Be careful at this stage. Some calculators will approximate the exponentiation (the calculator applications in Ubuntu 18.08 and Windows 10 do not, but older desk calculators will). You may try an arbitrary precision calculator, such as `bc` (see Chapter 3). The output from the exponentiation using `bc` is:

```
138^107
92696267009151974112580966494142469075148237762435797813883675229744\
10315603725576855575549455980054411733018856229158449793951447981059\
64058537231504845445105996494390906329961481123710256232656386293889\
61097155080260346099793 92
```

Then performing the mod gives:

```
138^107 % 187
8
```

Therefore user A has successfully decrypted the ciphertext, obtaining the original plaintext, $M = 8$.

13.2 Analysis of RSA

We now analyse the design of RSA, identifying appropriate values to be used that lead to suitable security and performance.

- Encryption involves taking plaintext and raise to power e
- Decryption involves taking previous value and raise to a different power d
- Decryption must produce the original plaintext, that is:

$$(M^e)^d \bmod n = M \text{ for all } M < n$$

- This is true of if e and d are relatively prime
- Choose primes p and q, and calculate:

$$\begin{aligned}
&n = pq\\
&1 < e < \phi(n)\\
&ed \equiv 1 \pmod{\phi(n)} \text{ or } d \equiv e^{-1} \pmod{\phi(n)}
\end{aligned}$$

Here we see why the key generation algorithm is designed as it is. Decryption will only work (that is, produce the original plaintext) if the top equation is true. Note that $M^{e^d} = M^{ed}$. So the condition is that if you take the plaintext M and raise it to the power ed then the answer must be the original M (in mod n). For this to be true, e and d must be chosen appropriately—it will not work for just any value of e and d. Using Euler's theorem it can be shown that it will be true if e and d are multiplicative inverses of each other in mod $\phi(n)$.

Now we consider the guidelines for choosing values of parameters in RSA key generation.

- Note: modular exponentiation is slow when using large values
- Choosing e
  - Values such as 3, 17 and 65537 are popular: make exponentiation faster
  - Small e vulnerable to attack; solution is to add random padding to each M
- Choosing d
  - Small d vulnerable to attack
  - But large d makes decryption slow
- Choosing p and q
  - p and q must be very large primes
  - Choose random odd number and test if its prime (probabilistic test)

As we saw in the exercise, key generation involves selecting values for p, q and e (where e influences the value of d as it is the multiplicative inverse).

As e is a public value, a small value can be selected (since a brute force is not relevant; the attacker already knows it) and in fact, many users can use the same value as each other. For example, OpenSSL defaults to using $e = 2^{16} + 1 = 65537$ for all keypairs generated. That is, by default everyone using OpenSSL to generate keypairs will have the same value of e. This value is small, meaning encryption is reasonable fast.

As d is the multiplicative inverse of e, a small e means d will be large. This is good, because d must be kept private; large values are not subject to brute force attack. But it makes decryption slow, since it involves M^d, which is often taking one very large number M and raising to the power of another very large number d. We will see later there are algorithms that can speed up the decryption process.

The primes p and q should be chosen randomly (again, they are private, so should be hard for an attacker to guess). A common approach is to choose a large odd number and then check if it is prime. There are primality testing algorithms that can either prove the number selected is prime, or give high confidence that it is prime (i.e. probabilistic test). When RSA is used for signatures—it's most common use—probabilistic testing is sufficient (it is faster than testing for provable primes).

Now we look at why RSA is considered secure by considering the possible attacks on RSA.

- Brute-Force attack: choose large d (but makes algorithm slower)
- Mathematical attacks:
  1. Factor n into its two prime factors
  2. Determine $\phi(n)$ directly, without determining p or q
  3. Determine d directly, without determining $\phi(n)$
- Factoring n is considered fastest approach; hence used as measure of RSA security
- Timing attacks: practical, but countermeasures easy to add (e.g. random delay). 2 to 10% performance penalty
- Chosen ciphertext attack: countermeasure is to use padding (Optimal Asymmetric Encryption Padding)

The three mathematical attacks require the attacker to solve computationally hard problems. That is, when large values are used,

Video
Analysis of RSA (6 min; Apr 2021)
https://www.youtube.com/watch?v=CX2-Crudguk

Video
Avenues of attack on RSA (10 min; Feb 2015)
https://www.youtube.com/watch?v=ywlzcE3eQzQ

- Factoring n into primes p and q is considered the easiest attack
- Some records by length of n:
  - 1991: 330 bits (100 digits)
  - 2003: 576 bits (174 digits)
  - 2005: 640 bits (193 digits)
  - 2009: 768 bits (232 digits), 10^{20} operations, 2000 years on single core 2.2 GHz computer
  - 2019: 795 bits (240 digits), 900 core years
- Improving at rate of 5–20 bits per year
- Typical length of n: 1024 bits, 2048 bits, 4096 bits

In the 1990's and 2000's, the RSA Challenge tasked researchers with factoring integers of various sizes. The numbers reported on this slide are mainly from successful attempts at the RSA Challenge.

The rate of improvement of integer factorisation, varies depending on where you consider the starting year. In any case, RSA keys of 2048 bits are considered secure for the near future.

We don't cover quantum computers and cryptography here. While it is important for the future, in 2018 the largest reported integer factored into primes using a quantum computer was 4088459, that is 22 bits. While in theory quantum computers will be able to make integer factorisation much easier (make RSA insecure), in practice there is a long way to go.

13.3 Implementations of RSA

We now consider implementation aspects of RSA, first looking at the recommend parameter values and then how implementations can improve performance.

- RSA Key length: 1024, 2048, 3072 or 4096 bits
  - Refers to the length of n
  - 2048 and above are recommended
- p and q are chosen randomly; about half as many bits as n
- e is small, often constant; e.g. 65537
- d is calculated; about same length as n
- For detailed recommendations see NIST FIPS 186 Digital Signature Standard

As an example, with a RSA 1024 bit key, length of p and q will be about 512 bits, and the length of n will be 1024 bits. e could be 65537 which is 17 bits, and d will be approximately 1024 bits.

FIPS 186 provides details of the implementation of RSA to meet US government standards. It includes specific algorithms to use and some recommended values. It also sets requirements for selecting random primes.

- Modular arithmetic, especially exponentiation, can be slow with very large numbers (1000's of bits)
- Use properties of modular arithmetic to simplify calculations, e.g.

$$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$$

- Also Euler's theorem and Chinese Remainder Theorem can simplify calculations
- Decryption is significantly slower than encryption since d is very large
- Implementations of RSA often store and use intermediate values to speed up decryption

While there are methods to speed up decryption in RSA (see the next slide), it is still significantly slower than encryption in practice.

- Encryption:

$$C = M^e \bmod n$$

- Decryption:

$$M = C^d \bmod n$$

- Modulus, n of length b bits
- Public exponent, e
- Private exponent, d
- Prime1, p, and Prime2, q
- Exponent1, $d_p = d \pmod{p-1}$
- Exponent2, $d_q = d \pmod{q-1}$
- Coefficient, $q_{inv} = q^{-1} \pmod{p}$
- Private values: $PR = \{n, e, d, p, q, d_p, d_q, q_{inv}\}$
- Public values: $PU = \{n, e\}$

We see the parameters used within OpenSSL. p, q, n, e and d are normal. However d_p, d_q and q_{inv} are intermediate values introduced and stored as part of the private key. They are used to speed up the decryption calculation. The decryption algorithm is split into multiple steps using these intermediate values, such that it is significant faster than if using a single step. However the end result is still the same.

While you don't need to know what the intermediate steps are, it is useful to know that these intermediate values exist, as you will see them when using RSA in practice (e.g. generating keys with OpenSSL).

13.4 RSA in OpenSSL

OpenSSL can be used to perform various operations with public key cryptography. Here we demonstrate basic usage of RSA.

To demonstrate RSA we use the scenario of user Alice on one computer (called `node1`) wishing to send a confidential and signed message to user Bob on another computer (called `node2`). This chapter has focused on using RSA for encryption (i.e. keeping the message confidential). However it is much more widely used for authentication or signing messages. This example includes signing; the concepts of signing and verification are discussed in Chapter 17.

We demonstrate the following operations for users:

- Create a RSA public/private key pair
- View and understand the parameters in the key pair
- Sign a message using their private key
- Encrypt a message using the recipients public key
- "Send" the signature and ciphertext to the recipient

13.4.1 RSA Key Generation in OpenSSL

Any user can generate their RSA key pair using the `genpkey` command. Note that in public key cryptography a key pair consists of a private key and public key. A user can distribute their public key to anyone, but keeps their private key to themselves. But they also need to store their own public key. So in practice, a user will have two files: a private key file, which contains their private key information *and* their public key information; and a public key file, which contains *only* their public key information. So in OpenSSL, when a private key is generated with `genpkey`, the public key information is also created.

To generate the private (and public key):

```
alice@node1:~$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048
    -pkeyopt rsa_keygen_pubexp:3 -out privkey-alice.pem
alice@node1:~$ cat privkey-alice.pem
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCoXEAmbAuh9Nks
xtjIqgW8+MjaoRLWIKOpr54E7XcpzMSlNZggPBp0sLjfgvNFBPP7BrQms3qigwow
krML/fdwSFybigmuTCyJS/UIn3J5s70vUSpQ9M8oAU+6lvRdiByqR0zBnnWdR9B8
wW2/jM2Ng3yq51S6qR6LUs92jEzYATz1df8z+qcUL+navmOSLdA110qQpbKjEjI1
esJIkqrKlQiu1N0TQbexC9dNwtI79G79UR+YOR8CWJyYy/ZPeUrsr1mcSGL7facW
/aG2hh85/XdICm2PWgRySUu0M2rHdxL+AMukauYnlw4gddTO0cmUNyxKrVr5aQBP
hZxKtFV5AgEDAoIBAHA9gBmdXRajO3MvOzBxWSil2zxrYeQVwnEfvq3zpMaIgxjO
ZWrSvE3LJepXTNit9/yvIsR3pxcCBssMd11T+kra6GexW8mIHbDdTgW/oaZ303Tg
xuCjNMVWNScPTZOwExwviIEUTmjaiv3WSSpd3l5XqHHvjdHGFFzh36RdiI//vcSX
VHC76AkhkJ13aDEIUSQPMfE0OmI4dgK2sxH8BXAmAgc7YOksLF4t+tjaEoeUFQWP
SwFiGgVaU3wtmv1DoSwbAKSWs/9hDg3vgN8AFku3HCdBkpmpp2CYqoBWFDfUNW2q
TtB7IU2fwUOtoqiW8CegqVNf+X+KWT85mb1NnqMCgYEA3z2IhWyENYsHRrfbpISR
q3y5l5sgFM1ofRbPA5AZbZANY48jFPSeuKWJ1HhhZpwai+dcKf5R2w5V/4vpKqec
wFFGkXiOshkzty/67A75Uww/iewff0nj8ZwG7oLYl2PHu7iyyHiwbTj7N21Rapq+
iUHpd4RBpiOPoad4lD+CDWcCgYEAwREKex5clXt2SjavosQPqwMG6Au3RkJVBBqZ
```

```
sh1/NRJOohTYtsDgvH49CpAaT9R7w42eBRfUHOv7H9KeYyv3GNlARyzXouM4WtIb
dFkMqrwrQyEIkl73l8VdXXDZtQ/xByDOjPMBxvosNM2f9jcw2BbctslbvpaJ2Mk2
oW892h8CgYEAlNOwWPMCzlyvhHqSba22clMmZRIVYzOa/g80rQq7nmAI7QoXY02/
JcOxOFBA7xK8XUToG/7hPLQ5VQfwxxpogDYvC6W0drt3z3VR8rSmN11/sUgU/4aX
9mgEnwHlukKFJ9B3MFB1niX8z542RxHUW4FGT62BGW0Ka8T7DX+sCO8CgYEAgLYG
/L7oY6ekMXnKbIK1HKyvRV0k2YGOArxmdr5Uzgw0bA3lzytAfal+Bwq8NThSgl5p
WLqNaJ1SFTcUQh1PZeYq2h3lF0IlkeFnouYIcdLHghYFtun6ZS4+Pks7zgqgr2s0
XfdWhKbIIzO/+XogkA89zzDn1GRb5dt5wPTT5r8CgYEA29235n/Hw7wzOJyao6nO
3rjCZon4/V2G800VJF5hhAqCX5KDLd0KIMbaHaxsjW+n79CqZSUz3kZtpSXBXRJ7
SIXoCYljaoxdJ6SkVED6uFmcZ+3iwioxXzpIFIW0ZZj5S/WgBkPsioAJ6Cp5S8zh
BFB15UA+JWFH2SRabjXf0+4=
-----END PRIVATE KEY-----
```

The **genpkey** command takes an algorithm (RSA) as an option, and that algorithm may have further specific options. In this example we set the RSA key length to 2048 bits and used a public exponent of 3. Omitting these -pkeyopt options will revert to the default values. The private key (and public key information) is output to a file.

The private key file is encoded with Base64. To view the values:

```
alice@node1:~$ openssl pkey -in privkey-alice.pem -text
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCoXEAmbAuh9Nks
xtjIqgW8+MjaoRLWIKOpr54E7XcpzMSlNZggPBp0sLjfgvNFBPP7BrQms3qigwow
krML/fdwSFybigmuTCyJS/UIn3J5s70vUSpQ9M8oAU+6lvRdiByqR0zBnnWdR9B8
wW2/jM2Ng3yq51S6qR6LUs92jEzYATz1df8z+qcUL+navmOSLdA110qQpbKjEjI1
esJIkqrKlQiu1N0TQbexC9dNwtI79G79UR+YOR8CWJyYy/ZPeUrsr1mcSGL7facW
/aG2hh85/XdICm2PWgRySUuOM2rHdxL+AMukauYnlw4gddTOOcmUNyxKrVr5aQBP
hZxKtFV5AgEDAoIBAHA9gBmdXRajO3MvOzBxWSil2zxrYeQVwnEfvq3zpMaIgxjO
ZWrSvE3LJepXTNit9/yvIsR3pxcCBssMd11T+kra6GexW8mIHbDdTgW/oaZ3O3Tg
xuCjNMVWNScPTZOwExwviIEUTmjaiv3WSSpd3l5XqHHvjdHGFFzh36RdiI//vcSX
VHC76AkhkJ13aDEIUSQPMfEOOmI4dgK2sxH8BXAmAgc7YOksLF4t+tjaEoeUFQWP
SwFiGgVaU3wtmv1DoSwbAKSWs/9hDg3vgN8AFku3HCdBkpmpp2CYqoBWFDfUNW2q
TtB7IU2fwUOtoqiW8CegqVNf+X+KWT85mb1NnqMCgYEA3z2IhWyENYsHRrfbpISR
q3y5l5sgFM1ofRbPA5AZbZANY48jFPSeuKWJ1HhhZpwai+dcKf5R2w5V/4vpKqec
wFFGkXiOshkzty/67A75Uww/iewff0nj8ZwG7oLYl2PHu7iyyHiwbTj7N21Rapq+
iUHpd4RBpiOPoad4lD+CDWcCgYEAwREKex5clXt2SjavosQPqwMG6Au3RkJVBBqZ
sh1/NRJOohTYtsDgvH49CpAaT9R7w42eBRfUHOv7H9KeYyv3GNlARyzXouM4WtIb
dFkMqrwrQyEIkl73l8VdXXDZtQ/xByDOjPMBxvosNM2f9jcw2BbctslbvpaJ2Mk2
oW892h8CgYEAlNOwWPMCzlyvhHqSba22clMmZRIVYzOa/g80rQq7nmAI7QoXY02/
JcOxOFBA7xK8XUToG/7hPLQ5VQfwxxpogDYvC6W0drt3z3VR8rSmN11/sUgU/4aX
9mgEnwHlukKFJ9B3MFB1niX8z542RxHUW4FGT62BGW0Ka8T7DX+sCO8CgYEAgLYG
/L7oY6ekMXnKbIK1HKyvRV0k2YGOArxmdr5Uzgw0bA3lzytAfal+Bwq8NThSgl5p
WLqNaJ1SFTcUQh1PZeYq2h3lF0IlkeFnouYIcdLHghYFtun6ZS4+Pks7zgqgr2s0
XfdWhKbIIzO/+XogkA89zzDn1GRb5dt5wPTT5r8CgYEA29235n/Hw7wzOJyao6nO
3rjCZon4/V2G800VJF5hhAqCX5KDLd0KIMbaHaxsjW+n79CqZSUz3kZtpSXBXRJ7
SIXoCYljaoxdJ6SkVED6uFmcZ+3iwioxXzpIFIW0ZZj5S/WgBkPsioAJ6Cp5S8zh
BFB15UA+JWFH2SRabjXf0+4=
-----END PRIVATE KEY-----
Private-Key: (2048 bit)
modulus:
    00:a8:5c:40:26:6c:0b:a1:f4:d9:2c:c6:d8:c8:aa:
    05:bc:f8:c8:da:a1:12:d6:20:a3:a9:af:9e:04:ed:
    77:29:cc:c4:a5:35:98:20:3c:1a:74:b0:b8:df:82:
    f3:45:04:f3:fb:06:b4:26:b3:7a:a2:83:0a:30:92:
    b3:0b:fd:f7:70:48:5c:9b:8a:09:ae:4c:2c:89:4b:
    f5:08:9f:72:79:b3:bd:2f:51:2a:50:f4:cf:28:01:
    4f:ba:96:f4:5d:88:1c:aa:47:4c:c1:9e:75:9d:47:
    d0:7c:c1:6d:bf:8c:cd:8d:83:7c:aa:e7:54:ba:a9:
    1e:8b:52:cf:76:8c:4c:d8:01:3c:f5:75:ff:33:fa:
    a7:14:2f:e9:da:be:63:92:2d:d0:35:d7:4a:90:a5:
    b2:a3:12:32:35:7a:c2:48:92:aa:ca:95:08:ae:d4:
    dd:13:41:b7:b1:0b:d7:4d:c2:d2:3b:f4:6e:fd:51:
    1f:98:39:1f:02:58:9c:98:cb:f6:4f:79:4a:ec:af:
    59:9c:48:62:fb:7d:a7:16:fd:a1:b6:86:1f:39:fd:
    77:48:0a:6d:8f:5a:04:72:49:4b:b4:33:6a:c7:77:
    12:fe:00:cb:a4:6a:e6:27:97:0e:20:75:d4:ce:d1:
    c9:94:37:2c:4a:ad:5a:f9:69:00:4f:85:9c:4a:b4:
    55:79
publicExponent: 3 (0x3)
privateExponent:
    70:3d:80:19:9d:5d:16:a3:3b:73:2f:3b:30:71:59:
    28:a5:db:3c:6b:61:e4:15:c2:71:1f:be:ad:f3:a4:
    c6:88:83:18:ce:65:6a:d2:bc:4d:cb:25:ea:57:4c:
    d8:ad:f7:fc:af:22:c4:77:a7:17:02:06:cb:0c:77:
    5d:53:fa:4a:da:e8:67:b1:5b:c9:88:1d:b0:dd:4e:
    05:bf:a1:a6:77:d3:74:e0:c6:e0:a3:34:c5:56:35:
    27:0f:4d:93:b0:13:1c:2f:88:81:14:4e:68:da:8a:
    fd:d6:49:2a:5d:de:5e:57:a8:71:ef:8d:d1:c6:14:
    5c:e1:df:a4:5d:88:8f:ff:bd:c4:97:54:70:bb:e8:
    09:21:90:9d:77:68:31:08:51:24:0f:31:f1:34:3a:
    62:38:76:02:b6:b3:11:fc:05:70:26:02:07:3b:60:
    e9:2c:2c:5e:2d:fa:d8:da:12:87:94:15:05:8f:4b:
    01:62:1a:05:5a:53:7c:2d:9a:fd:43:a1:2c:1b:00:
    a4:96:b3:ff:61:0e:0d:ef:80:df:00:16:4b:b7:1c:
    27:41:92:99:a9:a7:60:98:aa:80:56:14:37:d4:35:
    6d:aa:4e:d0:7b:21:4d:9f:c1:43:ad:a2:a8:96:f0:
    27:a0:a9:53:5f:f9:7f:8a:59:3f:39:99:bd:4d:9e:
    a3
prime1:
    00:df:3d:88:85:6c:84:35:8b:07:46:b7:db:a4:84:
    91:ab:7c:b9:97:9b:20:14:cd:68:7d:16:cf:03:90:
    19:6d:90:0d:63:8f:23:14:f4:9e:b8:a5:89:d4:78:
    61:66:9c:1a:8b:e7:5c:29:fe:51:db:0e:55:ff:8b:
    e9:2a:a7:9c:c0:51:46:91:78:8e:b2:19:33:b7:2f:
    fa:ec:0e:f9:53:0c:3f:89:ec:1f:7f:49:e3:f1:9c:
    06:ee:82:d8:97:63:c7:bb:b8:b2:c8:78:b0:6d:38:
    fb:37:6d:51:6a:9a:be:89:41:e9:77:84:41:a6:23:
    8f:a1:a7:78:94:3f:82:0d:67
prime2:
    00:c1:11:0a:7b:1e:5c:95:7b:76:4a:36:af:a2:c4:
    0f:ab:03:06:e8:0b:b7:46:42:55:04:1a:99:b2:1d:
    7f:35:12:4e:a2:14:d8:b6:c0:e0:bc:7e:3d:0a:90:
    1a:4f:d4:7b:c3:8d:9e:05:17:d4:1c:eb:fb:1f:d2:
    9e:63:2b:f7:18:d9:40:47:2c:d7:a2:e3:38:5a:d2:
    1b:74:59:0c:aa:bc:2b:43:21:08:92:5e:f7:97:c5:
    5d:5d:70:d9:b5:0f:f1:07:20:ce:8c:f3:01:c6:fa:
    2c:34:cd:9f:f6:37:30:d8:16:dc:b6:c9:5b:be:96:
    89:d8:c9:36:a1:6f:3d:da:1f
exponent1:
    00:94:d3:b0:58:f3:02:ce:5c:af:84:7a:92:6d:ad:
    b6:72:53:26:65:12:15:63:33:9a:fe:0f:34:ad:0a:
    bb:9e:60:08:ed:0a:17:63:4d:bf:25:c3:b1:38:50:
    40:ef:12:bc:5d:44:e8:1b:fe:e1:3c:b4:39:55:07:
```

```
    f0:c7:1a:68:80:36:2f:0b:a5:b4:76:bb:77:cf:75:
    51:f2:b4:a6:37:5d:7f:b1:48:14:ff:86:97:f6:68:
    04:9f:01:e5:ba:42:85:27:d0:77:30:50:75:9e:25:
    fc:cf:9e:36:47:11:d4:5b:81:46:4f:ad:81:19:6d:
    0a:6b:c4:fb:0d:7f:ac:08:ef
exponent2:
    00:80:b6:06:fc:be:e8:63:a7:a4:31:79:ca:6c:82:
    b5:1c:ac:af:45:5d:24:d9:81:8e:02:bc:66:76:be:
    54:ce:0c:34:6c:0d:e5:cf:2b:40:7d:a9:7e:07:0a:
    bc:35:38:52:82:5e:69:58:ba:8d:68:9d:52:15:37:
    14:42:1d:4f:65:e6:2a:da:1d:e5:17:42:25:91:e1:
    67:a2:e6:08:71:d2:c7:82:16:05:b6:e9:fa:65:2e:
    3e:3e:4b:3b:ce:0a:a0:af:6b:34:5d:f7:56:84:a6:
    c8:23:33:bf:f9:7a:20:90:0f:3d:cf:30:e7:d4:64:
    5b:e5:db:79:c0:f4:d3:e6:bf
coefficient:
    00:db:dd:b7:e6:7f:c7:c3:bc:33:38:9c:9a:a3:a9:
    ce:de:b8:c2:66:89:f8:fd:5d:86:f3:4d:15:24:5e:
    61:84:0a:82:5f:92:83:2d:dd:0a:20:c6:da:1d:ac:
    6c:8d:6f:a7:ef:d0:aa:65:25:33:de:46:6d:a5:25:
    c1:5d:12:7b:48:85:e8:09:89:63:6a:8c:5d:27:a4:
    a4:54:40:fa:b8:59:9c:67:ed:e2:c2:2a:31:5f:3a:
    48:14:85:b4:65:98:f9:4b:f5:a0:06:43:ec:8a:80:
    09:e8:2a:79:4b:cc:e1:04:50:75:e5:40:3e:25:61:
    47:d9:24:5a:6e:35:df:d3:ee
```

An explanation of these values can be found in a lecture on Public Key Cryptography, specifically on slide 18.

To output just the public key to a file:

```
alice@node1:~$ openssl pkey -in privkey-alice.pem -out pubkey-alice.pem -pubout
alice@node1:~$ cat pubkey-alice.pem
-----BEGIN PUBLIC KEY-----
MIIBIDANBgkqhkiG9w0BAQEFAAOCAQ0AMIIBCAKCAQEAqFxAJmwLofTZLMbYyKoF
vPjI2qES1iCjqa+eBO13KczEpTWYIDwadLC434LzRQTz+wa0JrN6ooMKMJKzC/33
cEhcm4oJrkwsiUv1CJ9yebO9L1EqUPTPKAFPupb0XYgcqkdMwZ51nUfQfMFtv4zN
jYN8qudUuqkei1LPdoxM2AE89XX/M/qnFC/p2r5jki3QNddKkKWyoxIyNXrCSJKq
ypUIrtTdE0G3sQvXTcLSO/Ru/VEfmDkfAlicmMv2T3lK7K9ZnEhi+32nFv2htoYf
Of13SAptj1oEcklLtDNqx3cS/gDLpGrmJ5cOIHXUztHJlDcsSq1a+WkAT4WcSrRV
eQIBAw==
-----END PUBLIC KEY-----
```

Check by looking at the individual values. Only the public key values are included:

```
alice@node1:~$ openssl pkey -in pubkey-alice.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIIBIDANBgkqhkiG9w0BAQEFAAOCAQ0AMIIBCAKCAQEAqFxAJmwLofTZLMbYyKoF
vPjI2qES1iCjqa+eBO13KczEpTWYIDwadLC434LzRQTz+wa0JrN6ooMKMJKzC/33
cEhcm4oJrkwsiUv1CJ9yebO9L1EqUPTPKAFPupb0XYgcqkdMwZ51nUfQfMFtv4zN
jYN8qudUuqkei1LPdoxM2AE89XX/M/qnFC/p2r5jki3QNddKkKWyoxIyNXrCSJKq
ypUIrtTdE0G3sQvXTcLSO/Ru/VEfmDkfAlicmMv2T3lK7K9ZnEhi+32nFv2htoYf
Of13SAptj1oEcklLtDNqx3cS/gDLpGrmJ5cOIHXUztHJlDcsSq1a+WkAT4WcSrRV
eQIBAw==
-----END PUBLIC KEY-----
Public-Key: (2048 bit)
Modulus:
    00:a8:5c:40:26:6c:0b:a1:f4:d9:2c:c6:d8:c8:aa:
    05:bc:f8:c8:da:a1:12:d6:20:a3:a9:af:9e:04:ed:
    77:29:cc:c4:a5:35:98:20:3c:1a:74:b0:b8:df:82:
    f3:45:04:f3:fb:06:b4:26:b3:7a:a2:83:0a:30:92:
    b3:0b:fd:f7:70:48:5c:9b:8a:09:ae:4c:2c:89:4b:
    f5:08:9f:72:79:b3:bd:2f:51:2a:50:f4:cf:28:01:
    4f:ba:96:f4:5d:88:1c:aa:47:4c:c1:9e:75:9d:47:
    d0:7c:c1:6d:bf:8c:cd:8d:83:7c:aa:e7:54:ba:a9:
    1e:8b:52:cf:76:8c:4c:d8:01:3c:f5:75:ff:33:fa:
    a7:14:2f:e9:da:be:63:92:2d:d0:35:d7:4a:90:a5:
    b2:a3:12:32:35:7a:c2:48:92:aa:ca:95:08:ae:d4:
    dd:13:41:b7:b1:0b:d7:4d:c2:d2:3b:f4:6e:fd:51:
    1f:98:39:1f:02:58:9c:98:cb:f6:4f:79:4a:ec:af:
    59:9c:48:62:fb:7d:a7:16:fd:a1:b6:86:1f:39:fd:
    77:48:0a:6d:8f:5a:04:72:49:4b:b4:33:6a:c7:77:
    12:fe:00:cb:a4:6a:e6:27:97:0e:20:75:d4:ce:d1:
    c9:94:37:2c:4a:ad:5a:f9:69:00:4f:85:9c:4a:b4:
    55:79
Exponent: 3 (0x3)
```

13.4.2 RSA Signing in OpenSSL (Sender)

Now that Alice has her private and public key files, let's create a text file containing the message to send to Bob:

```
alice@node1:~$ echo "This is my example message." > message-alice.txt
alice@node1:~$ cat message-alice.txt
This is my example message.
```

To sign the message you need to calculate its hash and then encrypt that hash using your private key. To create a hash of a message (without encrypting):

```
alice@node1:~$ openssl dgst -sha1 message-alice.txt
SHA1(message-alice.txt)= 064774b2fb550d8c1d7d39fa5ac5685e2f8b1ca6
```

OpenSSL has an option to calculate the hash and then sign it using a selected private key. The output will be a file containing the signature.

```
alice@node1:~$ openssl dgst -sha1 -sign privkey-alice.pem -out sign-alice.bin
    message-alice.txt
alice@node1:~$ ls -l
total 16
-rw-r--r-- 1 sgordon users 28 2012-03-04 15:14 message-alice.txt
-rw-r--r-- 1 sgordon users 1704 2012-03-04 14:58 privkey-alice.pem
-rw-r--r-- 1 sgordon users 451 2012-03-04 15:08 pubkey-alice.pem
-rw-r--r-- 1 sgordon users 256 2012-03-04 15:20 sign-alice.bin
```

13.4.3 RSA Encryption in OpenSSL (Sender)

To encrypt the message using RSA, use the recipients public key (this assumes the recipient, Bob, has already created and distributed their public key, using the same steps as above):

```
alice@node1:~$ openssl pkeyutl -encrypt -in message-alice.txt -pubin -inkey
    pubkey-bob.pem -out ciphertext-alice.bin
```

Note that direct RSA encryption should only be used on small files, with length less than the length of the key. If you want to encrypt large files then use symmetric key encryption. Two approaches to do this with OpenSSL: (1) generate a random key to be used with a symmetric cipher to encrypt the message and then encrypt the key with RSA; (2) use the `smime` operation, which combines RSA and a symmetric cipher to automate approach 1.

Now Alice sends the following to Bob:

- Ciphertext of the mesasge, `ciphertext-alice.bin`
- Signature of the message, `sign-alice.bin`
- Optionally, if she hasn't done so in the past, her public key, `public-alice.pem`

13.4.4 RSA Decryption in OpenSSL (Receiver)

When Bob receive's the two files from Alice, he needs to decrypt the ciphertext and verify the signature. Bob will need to use his RSA private/public key files, which were generated in the same way as for Alice, i.e. using `genpkey`.

To decrypt the received ciphertext:

```
bob@node2:~$ openssl pkeyutl -decrypt -in ciphertext-alice.bin -inkey
    privkey-bob.pem -out received-alice.txt
bob@node2:~$ cat received-alice.txt
This is my example message.
```

13.4.5 RSA Verification in OpenSSL (Receiver)

To verify the signature of a message:

```
bob@node2:~$ openssl dgst -sha1 -verify pubkey-alice.pem -signature
    sign-alice.bin received-alice.txt
Verified OK
```

The output messages shows the verification was successful.

13.4.6 RSA OpenSSL Exercises

Exercise 13.4 (RSA Key Generation)**.** Generate your own RSA key pair using the OpenSSL `genpkey` command. Extract your public key and then exchange public key's with another person (or if you want to do it on your own, generate a second key pair).

Solution 13.4 (RSA Key Generation)**.** See the examples of `genpkey` and `pkey` commands in Section 13.4.1.

Exercise 13.5 (RSA Signing)**.** Create a message in a file, sign that message using the `dgst` command, and then send the message and signature to another person.

Solution 13.5 (RSA Signing)**.** Use a text editor, such as `nano`, to create a file containing a message. See the examples of `dgst` in Section 13.4.2.

Exercise 13.6 (RSA Verification)**.** Verify the message you received.

Solution 13.6 (RSA Verification)**.** See the example in Section 13.4.5.

Exercise 13.7 (RSA Performance Test)**.** Using the OpenSSL `speed` command, compare the performance of RSA encrypt/sign operation against the RSA decrypt/verify operation.

Solution 13.7 (RSA Performance Test)**.** You can select the `rsa` algorithm using the `speed` command, so that the performance test is only for RSA (and doesn't include AES etc.).

13.5 RSA in Python

The Python Cryptography library includes asymmetric algorithms, including RSA. See the examples for RSA at:

- `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/`

Chapter 14

Diffie–Hellman Key Exchange

This chapter presents the Diffie–Hellman key exchange algorithm, which was the first example of public key cryptography.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

14.1 Diffie–Hellman Key Exchange Algorithm

Whitfield Diffie and Martin Hellman, along with Ralph Merkle, were the first to publish algorithms for public key cryptography. Their algorithm was designed to exchange a secret key between two parties, and commonly referred to as Diffie-Hellman Key Exchange (DHKE). You can read more about Diffie, Hellman and Merkle in Chapter C.

- Diffie and Hellman proposed public key cryptosystem in 1976
  - Motivation: solve the problem of how to exchange secret keys for symmetric key crypto
  - Proposed protocol for exchanging secrets using public keys
  - Merkle also contributed to the idea; sometimes called Diffie–Hellman-Merkle key exchange
- DHKE is algorithm for exchanging secret key (not for secrecy of data)
  - E.g. two users want to use symmetric key crypto, but need to first exchange a secret key
- Based on discrete logarithms
  - Easy to calculate exponential modulo a prime
  - Infeasible to calculate inverse, i.e. discrete logarithm

It is important to note that DHKE is a "key exchange" protocol. The purpose is for two users to exchange a secret key. Once a secret key has been exchanged with DHKE, the two users can then use that secret key for other purposes (e.g. for encrypting data using AES).

File: crypto/dh.tex, r1968

If you do not know what a discrete logarithm is, it is worth refreshing your knowledge in number theory from Chapter 5.

Algorithm 14.1 (Diffie–Hellman Key Exchange)**.** *One-time setup.* A and B agree upon public values prime p and generator g, where $g < p$ and g is a primitive root of p.

Protocol.

1. A: select private $PR_A < p$
2. A: calculate public $PU_A = g^{PR_A} \bmod p$
3. A → B: send PU_A
4. B: select private $PR_B < p$
5. B: calculate public $PU_B = g^{PR_B} \bmod p$
6. B: calculate secret $K_B = PU_A^{PR_B} \bmod p$
7. B → A: send PU_B
8. A: calculate secret $K_A = PU_B^{PR_A} \bmod p$

Result. $K_A = K_B$ is the shared secret value

The values p and g are either agreed upon in advance, or selected by one user and sent to the other in the first message. Both values are public; the attacker is assumed to know them.

When two users need to exchange a shared secret, one of them initiates the protocol. User A and B actually perform the same steps, but just with different values. First a private value PR is randomly selected. Then a public value PU is calculated. Both users exchange their public PU values (and the attacker may learn them). Finally, both users calculate their private values K based on their own PR and received PU. The values and calculations are designed such that the K calculated by each user will be the same. K is the shared secret key.

Note that the parameters have different variables or names in different sources. You may also see:

- prime p: q
- generator g: α
- private PR_A and PR_B: X_A and X_B; or a and b; or x and y
- public PU_A and PU_B: Y_A and Y_B; or A and B; or e and f
- secret K: s

Exercise 14.1 (Diffie–Hellman Key Exchange)**.** Assume two users, A and B, have agreed to use DHKE with prime $p = 19$ and generator $g = 10$. Assuming A randomly chose private $PR_A = 7$ and B randomly chose private $PR_B = 8$, find the shared secret key.

Solution 14.1 (Diffie–Hellman Key Exchange)**.** First note that $g = 10$ is in indeed a primitive root of $p = 19$, as seen in the examples on number theory in Section 5.4. That is, 10^0, 10^1, 10^2, 10^3, …, 10^{18} give distinct values in mod 19.

Let's consider the first phase from user A's perspective. A chooses private value $PR_A = 7$, which is less than 19. Then A calculates their public value:

$$\begin{aligned} PU_A &= g^{PR_A} \bmod p \\ &= 10^7 \bmod 19 \\ &= 10000000 \bmod 19 \\ &= 15 \end{aligned}$$

A sends $PU_A = 15$ to B.

Now consider from user B's perspective. B calculates their public value using their chosen $PR_B = 8$::

$$\begin{aligned} PU_B &= g^{PR_B} \bmod p \\ &= 10^8 \bmod 19 \\ &= 100000000 \bmod 19 \\ &= 17 \end{aligned}$$

B sends $PU_B = 17$ to A.

B can also calculate their version of the shared secret:

$$\begin{aligned} K_B &= PU_A^{PR_B} \bmod p \\ &= 15^8 \bmod 19 \\ &= 2562890625 \bmod 19 \\ &= 5 \end{aligned}$$

As A has received B's public value, A can also calculate their version of the shared secret:

$$\begin{aligned} K_A &= PU_B^{PR_A} \bmod p \\ &= 17^7 \bmod 19 \\ &= 410338673 \bmod 19 \\ &= 5 \end{aligned}$$

In summary, A and B have exchanged public values and then calculated a shared secret key of $K = 5$. Figure 14.1 illustrates the DHKE steps.

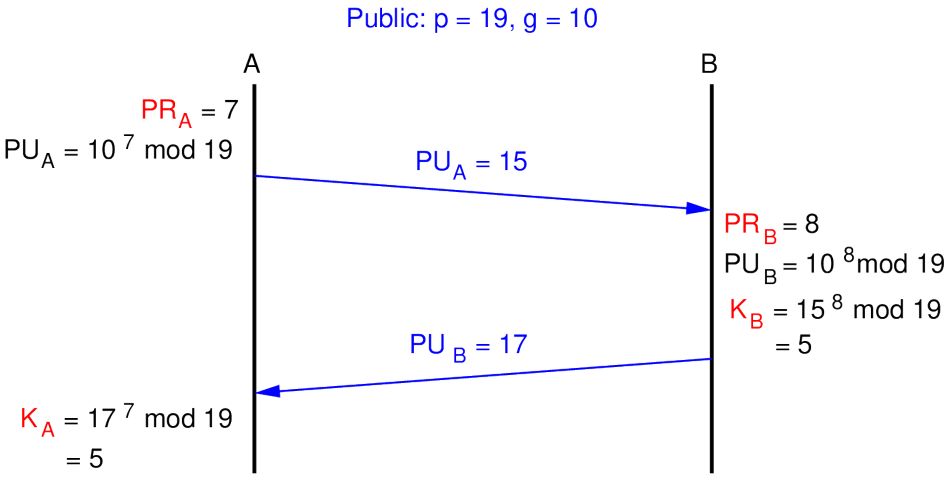


Figure 14.1: Diffie–Hellman Key Exchange Example

Video
Diffie-Hellman Key Exchange with Example (18 min; Mar 2015)
https://www.youtube.com/watch?v=p_UhlXDOlfU

14.2 Analysis of DHKE

1. Same shared secret: K_A and K_B must be identical
2. Computational efficiency: Easy to calculate PU and K
3. Secure: Infeasible to determine PR or K from known values
    - Attacker knows 3 public values in $PU_A = g^{PR_A} \bmod p$
    - Must be practically impossible to find the 4th value PR_A

While we don't show it here, it can easily be proved that DHKE will produce the same value of K for both users.

Modular exponentiation, while slow with big numbers, is easy to calculate, i.e. can be achieved in less than seconds.

The inverse operation of modular exponentiation, referred to as a discrete logarithm, is hard to calculate. With large enough values, it is considered impossible to calculate.

Question 14.1 (Prove Identical Keys in DHKE)**.** Prove that user A and user B will always calculate the same shared secret key in DHKE. That is, prove that $K_A = K_B$.

Video
Proof of Identical Keys in DHKE (5 min; Mar 2015
https://www.youtube.com/watch?v=y5G8YMA_sDU

Question 14.2 (Brute Force Attack on PR in DHKE)**.** Assuming you have intercepted $PU_A = 15$ from the DHKE exercise, how would you perform a brute force attack to find PR_A? How could such a successful brute force attack be prevented in practice?

Exercise 14.2 (Discrete Logarithm Attack in DHKE)**.** Assuming a brute force attack is not possible, write an equation that the attacker would have to solve to find PR_A.

Solution 14.2 (Discrete Logarithm Attack in DHKE)**.** Consider the equation:

$$PU_A = g^{PR_A} \bmod p$$

There is one unknown variable in the equation of four variables. The equation consists of modular exponentiation. The inverse operation is modular logarithm, or more commonly discrete logarithm, which can be written as:

$$PR_A = \mathrm{dlog}_{g,p}(PU_A)$$

which can be read as "given the base g and modulus p, find the index (or exponent) PR_A that produces the result PU_A".

- Discrete Logarithm Problem:

$$\text{given } g, p \text{ and } g^x \bmod p, \text{ find } x$$

- For certain values of p, considered computationally hard
  - p is a safe prime, i.e. $p = 2q + 1$ where q is a large prime
  - p is very large, usually at least 1024 bits
- 2016: Discrete logarithm with 768 bit prime p was solved within 5300 core years on 2.2GHz Xeon E5-2660 processor
- Considered harder to solve than equivalent integer factorisation
  - 768 bit integer factored in 2000 core years

14.3 Man-in-the-Middle Attack on DHKE

A very practical attack on DHKE is a Man-in-the-Middle (MITM) attack. If an attacker has the ability to intercept and send messages in between the two users, and the messages have no form of authentication, this attack can be successful.

Exercise 14.3 (MITM Attack on DHKE)**.** Consider the "Diffie–Hellman Key Exchange" exercise where user A chooses $PR_A = 7$ and B chooses $PR_B = 8$. Show how a MITM can be performed such that an attacker Q can decrypt any communications between A and B that use the secret shared between A and B.

Solution 14.3 (MITM Attack on DHKE)**.** In a MITM attack, the attacker Q intercepts messages between A and B, and masquerades as A to B, and as B to A. So when A sends its public value PU_A to B, it is intercepted by Q. Q then masquerades as B: selecting it's own PR_{QA}, calculating a PU_{QA} and sending back to A. A and Q calculate a shared secret key which will be identical. Without authentication of messages, A thinks it is communicating with B (since it send a message to B, and received a reply from who they think is B).

Q then performs a DHKE with B, and B thinks this is with A. The end result is that A and Q have a shared secret, and B and Q have another shared secret, and both A and B think their shared secret is with each other.

Figure 14.2 illustrates the MITM, where Q chooses random private value $PR_{QA} = 4$ for the DHKE with A and $PR_{QB} = 12$ for the DHKE with B.

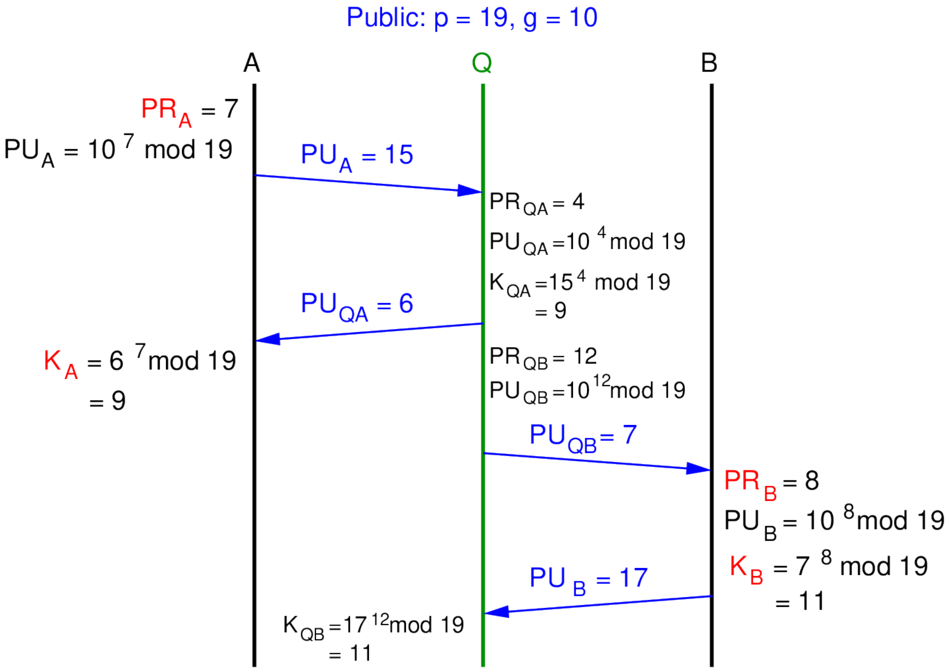


Figure 14.2: Diffie–Hellman MITM Attack Example

Now assuming the shared secrets are used as a key in a symmetric key cipher. When A encrypts a message and sends to B, Q can intercept and decrypt, since Q knows A's shared secret (9). Q can then encrypt the message with B's shared secret (11) and send on to B. B receives and decrypts, and subsequently responds to A. The exchange of encrypted data continues between A and B, without them noticing that Q is intercepting and decrypting the data.

Video
Man-in-the-Middle Attack on Diffie-Hellman Key Exchange (16 min; Mar 2015)
https://www.youtube.com/watch?v=Jokkhl8kq4c

14.4 Implementations of DHKE

- Some (older) communication protocols defined a fixed value of p and g
  - All clients and servers use the same values
- Newer protocols allow for an exchange of values (e.g. a Group Exchange protocol)
- Example fixed value in older versions of SSH (diffie-hellman-group1-sha1 using Oakley Group 2)

$$p = 2^{1024} - 2^{960} - 1 + 2^{64} \times (2^{894} \times \pi + 129093)$$

$$g = 2$$

  p is 1024 bits in length

As p and q are public and known to the attacker, using the same values all the time should not be a problem. Exchanging values involves extra communication overhead and also processing overhead. However following the principle of changing keys frequently to give an attacker less chance to compromise them, many protocols now support the ability to change the public parameters.

14.5 Diffie–Hellman in OpenSSL

OpenSSL supports the generation of public parameters for DHKE, as well as of the public and private keys of the users. The calculation of the secret key is also supported, although the public keys need to be manually exchanged. The main operations with OpenSSL are:

- Generate global public parameters with `genpkey`
- View parameters with `pkeyparam`
- Generate a public/private key pair with `genpkey`
- Extract a public key with `pkey`
- Derive a shared secret with `pkeyutl`

For this demo, we use the scenario of user Alice on `node1` and Bob on `node2`. Take note of the prompt to see who is performing each command.

The first step is to generate the Diffie-Hellman (DH) global public parameters, saving them in the file `dhp.pem`. We use the OpenSSL `genpkey` command, using the algorithm DH and the `-genparam` option:

```
alice@node1:~$ openssl genpkey -genparam -algorithm DH -out dhparam.pem
...+.......................................................................
.............................................................+.............
...............................................+...........................
...........................+........+......................................
.....................................+.....................................
...........................+..................................+........+...
...............+..........+...............+................................
```

```
.......................................................................+..
.................................................................+........
.............................................+........................+...
.....++*++*++*
```

Now let's display the generated global public parameters, first in the encoded form, then in the text form:

```
alice@node1:~$ cat dhparam.pem
-----BEGIN DH PARAMETERS-----
MIGHAoGBAOZVzJ4E8766527Mp3FD71xEUYdmFan4tPcSuPO99H7n9xfAm7WytmRQ
gxNn2dz4X58FKLzVMY+x2rLyPOd8SLa3OB7tE+gKFMymswteN//lPbFeLWtyei78
7lGJNnjVDpqJFmo1nldMTDyl5Z+ueZJP5vGGs2ouvem/Cf5N5QRTAgEC
-----END DH PARAMETERS-----
```

```
alice@node1:~$ openssl pkeyparam -in dhparam.pem -text
-----BEGIN DH PARAMETERS-----
MIGHAoGBAOZVzJ4E8766527Mp3FD71xEUYdmFan4tPcSuPO99H7n9xfAm7WytmRQ
gxNn2dz4X58FKLzVMY+x2rLyPOd8SLa3OB7tE+gKFMymswteN//lPbFeLWtyei78
7lGJNnjVDpqJFmo1nldMTDyl5Z+ueZJP5vGGs2ouvem/Cf5N5QRTAgEC
-----END DH PARAMETERS-----
PKCS#3 DH Parameters: (1024 bit)
    prime:
        00:e6:55:cc:9e:04:f3:be:ba:e7:6e:cc:a7:71:43:
        ef:5c:44:51:87:66:15:a9:f8:b4:f7:12:b8:f3:bd:
        f4:7e:e7:f7:17:c0:9b:b5:b2:b6:64:50:83:13:67:
        d9:dc:f8:5f:9f:05:28:bc:d5:31:8f:b1:da:b2:f2:
        3c:e7:7c:48:b6:b7:38:1e:ed:13:e8:0a:14:cc:a6:
        b3:0b:5e:37:ff:e5:3d:b1:5e:2d:6b:72:7a:2e:fc:
        ee:51:89:36:78:d5:0e:9a:89:16:6a:35:9e:57:4c:
        4c:3c:a5:e5:9f:ae:79:92:4f:e6:f1:86:b3:6a:2e:
        bd:e9:bf:09:fe:4d:e5:04:53
    generator: 2 (0x2)
```

Each user can use the public parameters to generate their own private and public key, saving them in their respective files. Similar to RSA, the DH private key file also stores the public key information.

```
alice@node1:~$ openssl genpkey -paramfile dhparam.pem -out dhprivkey-alice.pem
```

```
alice@node1:~$ openssl pkey -in dhprivkey-alice.pem -text -noout
PKCS#3 DH Private-Key: (1024 bit)
    private-key:
        48:88:7d:fd:09:0d:17:5e:33:be:ea:29:e7:b3:83:
        34:29:92:89:06:9f:9a:b4:92:b6:78:07:90:5f:aa:
        98:d9:6d:22:d7:92:05:be:f0:3f:14:af:09:3f:17:
        97:b9:04:73:41:32:c3:4a:38:8f:dc:79:e2:04:97:
        bf:a1:46:5f:ec:2a:ac:4f:ab:df:3b:b0:c9:be:86:
        85:d2:0f:7b:fe:03:46:a9:ab:df:7f:a8:98:38:c3:
        fa:9c:a6:ab:db:70:be:a6:67:95:ab:66:99:cc:15:
        4d:b5:94:90:e4:15:9f:14:2f:7b:dd:ff:60:3c:1d:
        3d:6c:4f:ff:81:77:e1:1d
    public-key:
        00:d9:ab:d7:8c:93:df:dd:eb:92:0d:57:d6:51:31:
```

```
        26:d8:f1:11:8c:92:37:a4:51:01:40:8d:bf:fe:6c:
        fd:95:b0:11:a0:16:e4:e0:ab:8a:ef:06:01:e8:36:
        a4:52:b8:bb:88:be:7c:a7:1e:4f:22:f9:7a:a6:5f:
        83:58:ee:69:34:8d:12:27:d6:5d:b6:e5:36:41:d1:
        a6:54:2a:a4:be:4b:4a:dc:75:fa:c8:16:af:79:a8:
        e3:f5:09:7f:83:13:e7:b7:25:df:37:ea:dc:8c:77:
        4e:20:33:df:a9:9c:95:cc:ef:33:3b:f4:02:b0:66:
        19:8c:30:48:1e:2a:83:87:5c
    prime:
        00:e6:55:cc:9e:04:f3:be:ba:e7:6e:cc:a7:71:43:
        ef:5c:44:51:87:66:15:a9:f8:b4:f7:12:b8:f3:bd:
        f4:7e:e7:f7:17:c0:9b:b5:b2:b6:64:50:83:13:67:
        d9:dc:f8:5f:9f:05:28:bc:d5:31:8f:b1:da:b2:f2:
        3c:e7:7c:48:b6:b7:38:1e:ed:13:e8:0a:14:cc:a6:
        b3:0b:5e:37:ff:e5:3d:b1:5e:2d:6b:72:7a:2e:fc:
        ee:51:89:36:78:d5:0e:9a:89:16:6a:35:9e:57:4c:
        4c:3c:a5:e5:9f:ae:79:92:4f:e6:f1:86:b3:6a:2e:
        bd:e9:bf:09:fe:4d:e5:04:53
    generator: 2 (0x2)
```

The other user uses the same public parameters, dhparam.pem, to generate their private/public key:

```
bob@node2:~$ openssl genpkey -paramfile dhparam.pem -out dhprivkey-bob.pem
```

```
bob@node2:~$ openssl pkey -in dhprivkey-bob.pem -text -noout
PKCS#3 DH Private-Key: (1024 bit)
    private-key:
        5d:70:9b:3e:a7:c9:b1:3b:df:17:d3:76:dd:45:f0:
        38:6d:be:35:f6:79:5d:05:bf:e2:63:b0:ea:25:00:
        61:0a:4c:e2:e4:e7:8e:97:6e:cb:9e:f0:f9:4b:d9:
        1c:2e:d6:b1:71:cb:ec:56:a7:2f:b0:af:ff:67:df:
        37:e0:d8:8c:ab:5d:ef:3d:27:c5:5a:a6:8d:49:30:
        6b:4e:d4:1f:5c:40:da:35:d0:bc:c7:3d:16:a3:13:
        2e:86:af:13:8b:65:c4:19:f2:75:43:e7:11:b6:5a:
        81:d1:e0:ff:5d:f3:c2:f4:6f:d2:f0:72:97:66:b9:
        93:3d:17:b0:06:ef:8a:3b
    public-key:
        00:d9:9a:00:1b:98:f5:0b:e2:d6:57:f7:4d:e3:4b:
        aa:43:ad:e2:f2:93:31:a1:e7:4b:a7:06:dc:ab:22:
        09:5a:0d:41:1a:c1:37:c0:6d:88:f4:7c:0a:22:27:
        1e:d3:84:39:51:92:62:d5:14:9e:68:ee:2f:69:27:
        ae:dd:d1:e6:a2:5f:3c:d2:7b:a7:7c:8e:61:28:fb:
        8b:1c:d7:a0:0b:d3:7b:37:af:78:b2:7e:eb:62:a7:
        85:b6:0f:90:10:b7:9c:ce:ec:84:a9:28:e3:7f:22:
        8f:76:cd:68:58:56:45:fd:3e:36:37:a1:99:aa:ca:
        4a:65:65:af:a8:21:ee:1f:b6
    prime:
        00:e6:55:cc:9e:04:f3:be:ba:e7:6e:cc:a7:71:43:
        ef:5c:44:51:87:66:15:a9:f8:b4:f7:12:b8:f3:bd:
        f4:7e:e7:f7:17:c0:9b:b5:b2:b6:64:50:83:13:67:
        d9:dc:f8:5f:9f:05:28:bc:d5:31:8f:b1:da:b2:f2:
        3c:e7:7c:48:b6:b7:38:1e:ed:13:e8:0a:14:cc:a6:
        b3:0b:5e:37:ff:e5:3d:b1:5e:2d:6b:72:7a:2e:fc:
        ee:51:89:36:78:d5:0e:9a:89:16:6a:35:9e:57:4c:
```

```
        4c:3c:a5:e5:9f:ae:79:92:4f:e6:f1:86:b3:6a:2e:
        bd:e9:bf:09:fe:4d:e5:04:53
    generator: 2 (0x2)
```

The users must exchange their public keys. To do so, they must first extract their public keys into separate files using the `pkey` command

```
alice@node1:~$ openssl pkey -in dhprivkey-alice.pem -pubout -out dhpub-alice.pem
```

Bob would perform a similar command as above with his keys (not shown).

We can view the public keys:

```
alice@node1:~$ openssl pkey -pubin -in dhpub-alice.pem -text
-----BEGIN PUBLIC KEY-----
MIIBIDCBlQYJKoZIhvcNAQMBMIGHAoGBAOZVzJ4E8766527Mp3FD71xEUYdmFan4
tPcSuPO99H7n9xfAm7WytmRQgxNn2dz4X58FKLzVMY+x2rLyPOd8SLa3OB7tE+gK
FMymswteN//lPbFeLWtyei787lGJNnjVDpqJFmo1nldMTDyl5Z+ueZJP5vGGs2ou
vem/Cf5N5QRTAgECA4GFAAKBgQDZq9eMk9/d65INV9ZRMSbY8RGMkjekUQFAjb/+
bP2VsBGgFuTgq4rvBgHoNqRSuLuIvnynHk8i+XqmX4NY7mk0jRIn1l225TZB0aZU
KqS+S0rcdfrIFq95qOP1CX+DE+e3Jd836tyMd04gM9+pnJXM7zM79AKwZhmMMEge
KoOHXA==
-----END PUBLIC KEY-----
PKCS#3 DH Public-Key: (1024 bit)
    public-key:
        00:d9:ab:d7:8c:93:df:dd:eb:92:0d:57:d6:51:31:
        26:d8:f1:11:8c:92:37:a4:51:01:40:8d:bf:fe:6c:
        fd:95:b0:11:a0:16:e4:e0:ab:8a:ef:06:01:e8:36:
        a4:52:b8:bb:88:be:7c:a7:1e:4f:22:f9:7a:a6:5f:
        83:58:ee:69:34:8d:12:27:d6:5d:b6:e5:36:41:d1:
        a6:54:2a:a4:be:4b:4a:dc:75:fa:c8:16:af:79:a8:
        e3:f5:09:7f:83:13:e7:b7:25:df:37:ea:dc:8c:77:
        4e:20:33:df:a9:9c:95:cc:ef:33:3b:f4:02:b0:66:
        19:8c:30:48:1e:2a:83:87:5c
    prime:
        00:e6:55:cc:9e:04:f3:be:ba:e7:6e:cc:a7:71:43:
        ef:5c:44:51:87:66:15:a9:f8:b4:f7:12:b8:f3:bd:
        f4:7e:e7:f7:17:c0:9b:b5:b2:b6:64:50:83:13:67:
        d9:dc:f8:5f:9f:05:28:bc:d5:31:8f:b1:da:b2:f2:
        3c:e7:7c:48:b6:b7:38:1e:ed:13:e8:0a:14:cc:a6:
        b3:0b:5e:37:ff:e5:3d:b1:5e:2d:6b:72:7a:2e:fc:
        ee:51:89:36:78:d5:0e:9a:89:16:6a:35:9e:57:4c:
        4c:3c:a5:e5:9f:ae:79:92:4f:e6:f1:86:b3:6a:2e:
        bd:e9:bf:09:fe:4d:e5:04:53
    generator: 2 (0x2)
```

After exchanging public keys, i.e. the files `dhpub-alice.pem` and `dhpub-bob.pem`, each user can derive the shared secret. Alice uses her private key and Bob's public key to derive a secret, in this case a 128 Byte binary value written into the file `secret-alice.bin`:

```
alice@node1:~$ openssl pkeyutl -derive -inkey dhprivkey-alice.pem -peerkey
    dhpubkey-bob.pem -out secret-alice.bin
```

Bob does the same using his private key and Alice's public key to produce his secret in the file `secret-bob.bin`:

```
bob@node2:~$ openssl pkeyutl -derive -inkey dhprivkey-bob.pem -peerkey
    dhpub-alice.pem -out secret-bob.bin
```

The secrets should be the same. Although there is no need for Bob to send his secret file to Alice, if he did, then Alice can use `cmp` to compare the files, or even `xxd` to manually inspect the binary values:

```
alice@node1:~$ cmp secret-alice.bin secret-bob.bin
alice@node1:~$ xxd secret-alice.bin
0000000: b7cb b892 b541 7810 d8ec d089 6c89 3c19 .....Ax.....l.<.
0000010: e8e1 27d8 66ee dac8 684a f0bd 0a7f e7d3 ..'.f...hJ......
0000020:␣3643␣8654␣fddf␣4399␣e58e␣2c7c␣3d33␣9532␣␣6C.T..C...,|=3.2
0000030:␣f693␣edf2␣c9a0␣40e8␣58b8␣38de␣74a5␣c0b0␣␣......@.X.8.t...
0000040:␣64ab␣4006␣a3cd␣d795␣2cef␣d0fc␣2b0f␣d1ab␣␣d.@.....,...+...
0000050:␣d1e5␣1a2a␣3431␣e3fa␣ba63␣f7cf␣1c61␣ff65␣␣...*41...c...a.e
0000060:␣d9cd␣c85d␣c5fe␣5c50␣c543␣aaeb␣de49␣8501␣␣...]..\P.C...I..
0000070:␣6cf1␣66a6␣87b6␣ddec␣835c␣b4b1␣3d9d␣e2fe␣␣l.f......\..=...
alice@node1:~$␣<kbd>xxd␣secret-bob.bin</kbd>
0000000:␣b7cb␣b892␣b541␣7810␣d8ec␣d089␣6c89␣3c19␣␣.....Ax.....l.<.
0000010:␣e8e1␣27d8␣66ee␣dac8␣684a␣f0bd␣0a7f␣e7d3␣␣..'.f...hJ......
0000020: 3643 8654 fddf 4399 e58e 2c7c 3d33 9532 6C.T..C...,|=3.2
0000030: f693 edf2 c9a0 40e8 58b8 38de 74a5 c0b0 ......@.X.8.t...
0000040: 64ab 4006 a3cd d795 2cef d0fc 2b0f d1ab d.@.....,...+...
0000050: d1e5 1a2a 3431 e3fa ba63 f7cf 1c61 ff65 ...*41...c...a.e
0000060: d9cd c85d c5fe 5c50 c543 aaeb de49 8501 ...]..\P.C...I..
0000070: 6cf1 66a6 87b6 ddec 835c b4b1 3d9d e2fe l.f......\..=...
```

Now both Alice and Bob have a shared secret, securely exchanged across a public network using DHKE.

14.6 DHKE in Python

The Python Cryptography library includes asymmetric algorithms, including RSA. See the examples for DHKE at:

- https://cryptography.io/en/latest/hazmat/primitives/asymmetric/

Chapter 15

Elliptic Curve Cryptography

RSA (Chapter 13) and Diffie-Hellman (Chapter 14) are two widely-used public key cryptography algorithms. Their security depends on the difficulty of factoring large integers into primes and solving discrete logarithms for integers, respectively. Their problem however is that keys are relatively large (e.g. 2048-bits for RSA). This leads to high communications overhead when exchanging keys in security protocols, and possibly performance limitations when implementing on low-cost computers.

Elliptic Curve Cryptography (ECC) is another, newer approach to public key cryptography. Mathematical operations are performed on an elliptic curve, where some operations can be easy if certain values are known, but practically impossible of those values are unknown. This is similar to the integer factorisation and discrete logarithm problems that make RSA and Diffie-Hellman secure. In fact, the problem is solving a discrete logarithm on an elliptic curve (rather than for integers as in Diffie-Hellman).

The main benefit of ECC is in performance. Specifically to achieve similar level of security as RSA and Diffie-Hellman, ECC has much smaller key sizes: 100's of bits vs 1000's of bits. Chapter 18 gives common recommended key lengths for RSA, Diffie-Hellman and ECC. In the past, RSA and (normal) Diffie-Hellman were favoured as ECC was relatively new. But now ECC is used in many applications, e.g. secret key exchange regularly uses the elliptic curve form of Diffie-Hellman rather than the normal, integer-based form[1].

This chapter gives a brief, as simple-as-possible, introduction to ECC.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

15.1 Overview of Elliptic Curve Cryptography

This section steps through the maths of elliptic curves, and explains why operations on an elliptic curve can be used for public key cryptography.

Definition 15.1 (Elliptic Curve)**.** An elliptic curve is defined by:

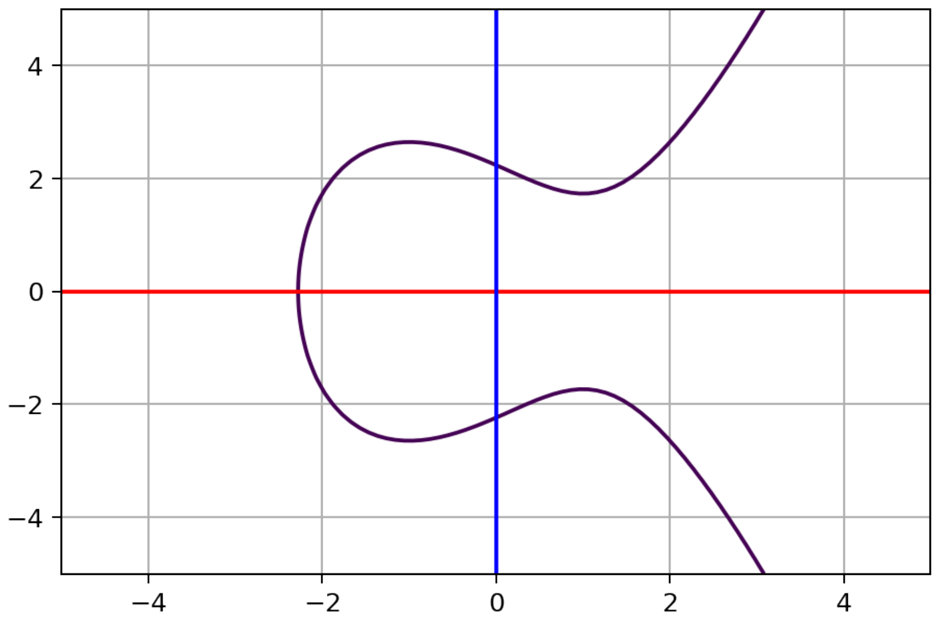
$$y^2 = x^3 + ax + b$$

File: crypto/elliptic.tex, r1949

[1] When referring to "Diffie-Hellman", we normally mean the algorithm based on integer discrete logarithms. However there is a Diffie-Hellman algorithm based on elliptic curve discrete logarithms. We will refer to this as "ECDH".

(with some constraints of constants a and b)

The constraints on a and b specify the relationship between the values, i.e. you cannot necessarily choose any values. We will not go into that detail here.



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

Figure 15.1: Elliptic Curve for $y^2 = x^3 - 3x + 5$

Figure 15.1 shows an example elliptic curve where $a = -3$ and $b = 5$, plotted for x values from -4 to 4. An elliptic curve always mirrors itself about the horizontal (red) axis.

Definition 15.2 (Addition Operation with an Elliptic Curve)**.** Select two points on the curve, A and B, and draw a straight line through them. The line will intersect with the curve at a third point, R (and no other points). The horizontal inverse of point R, is defined as the addition of A and B.

$$A + B = -R$$

See the following figure for an example of this concept. Note the points, A, B, R and -R are just (x, y) coordinates.

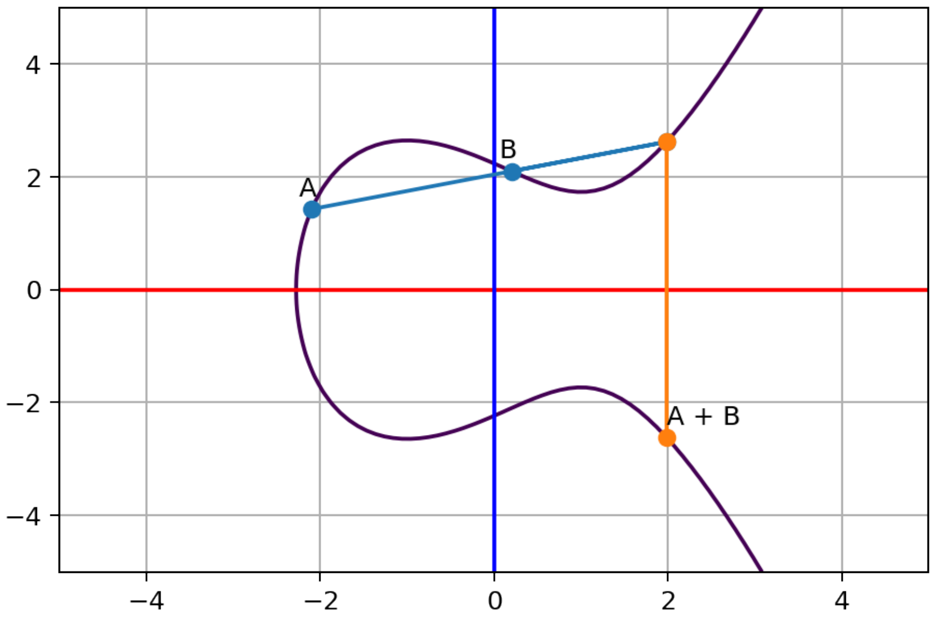
Figure 15.2 shows the concept of addition. Adding the points A and B results in the point shown as A+B. There is always a third point that intersects the curve on the line between A and B, and there is always an inverse of this point.

Note that we could continue the addition. For example, with A+B, add another point C, to arrive at a new point A+B+C. And so on.

Rather than adding two different points, we can simply add a single point to itself. The same concepts apply.

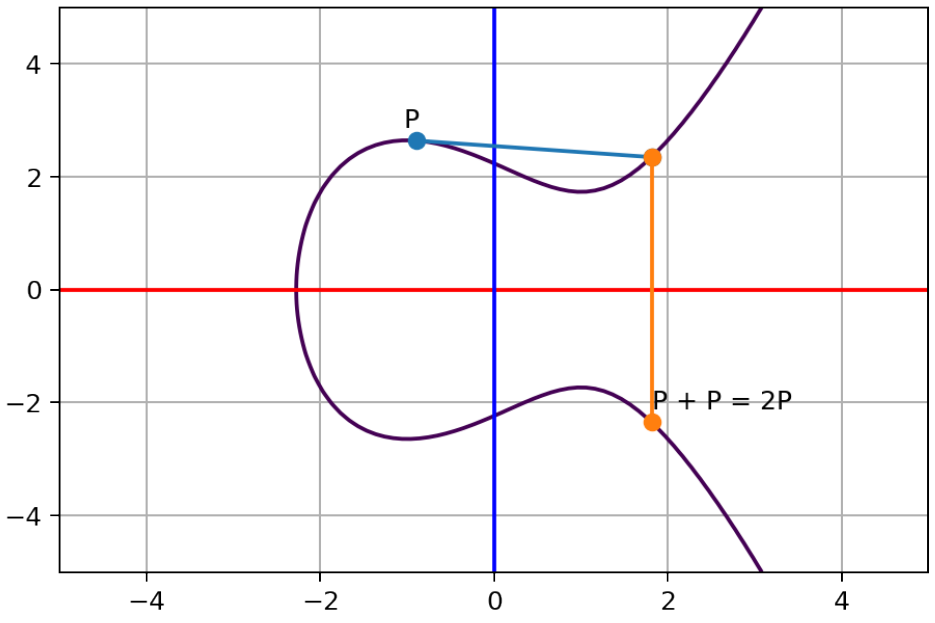
Figure 15.3 shows the self addition of point P. When adding a single point P to itself, the line that intersects P is chosen as the line tangent to P. So P+P = 2P.

We can continue to add P.



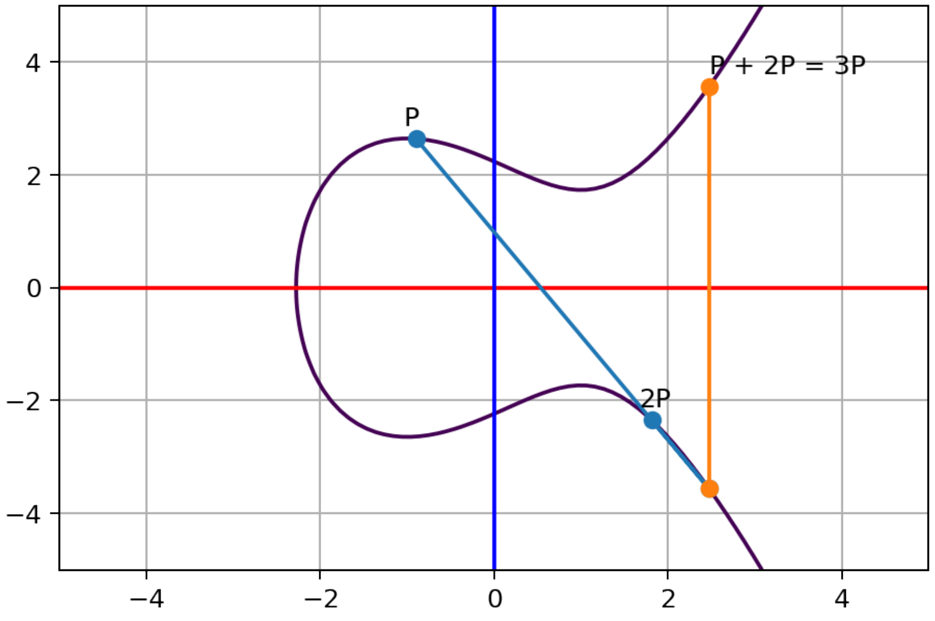
Credit: Generated based on MIT Licensed code by Fang-Pen Lin

Figure 15.2: Addition Operation on Elliptic Curve



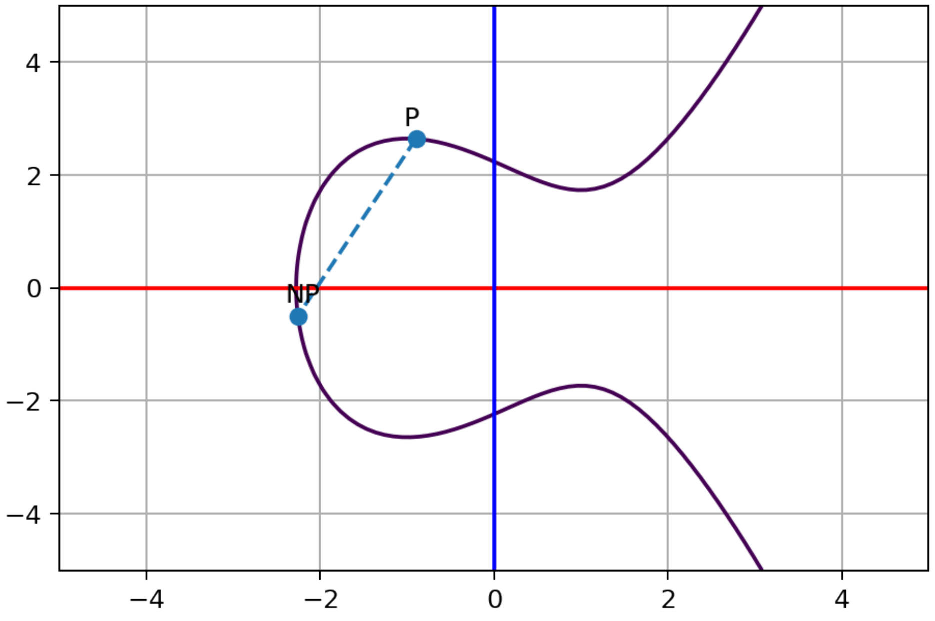
Credit: Generated based on MIT Licensed code by Fang-Pen Lin

Figure 15.3: Self Addition on Elliptic Curve



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

Figure 15.4: P + 2P on Elliptic Curve



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

Figure 15.5: NP on Elliptic Curve

Figure 15.4 shows P + 2P = 3P. Then we can add P again to get 4P and so on.

Figure 15.5 shows NP. In this example N=13. That is, we start with point P, and add P twelve times, resulting in the point 13P.

So now we know the concept of point addition on an elliptic curve, how can that be used for cryptography?

- User chooses a point P (global public parameter)
- User chooses a large, random N (private key)
- User calculates NP (public key)
  - Easy, since there is a shortcut (described shortly)
- Challenge for attacker: given NP, find N
  - Computationally hard for large N

As with other public key systems, elliptic curve cryptography relies on the fact that it is easy for the user to generate the public and private key, but practically impossible for an attacker to find the private key from the public key.

Why is that the case? So far we said NP is found by adding P $N-1$ times, that is, takes $N-1$ addition operations. So an attacker could simply start with P, and keep adding P until they get an answer of NP. Now the know how many additions, i.e. the private value N.

However if N is large enough the attackers method will be practically impossible. And for the user to generate NP when they know N, there is a shortcut that is practically achievable.

- Assume N is large, e.g. 256-bit random number
- Naive point addition: $P + P + P + P + \ldots + P + P$ ($2^{256} - 1$ additions)
- Shortcut algorithm for point addition:
  - Calculate P, $P+P = 2P = 2^1P$, $2P+2P = 4P = 2^2P$, $4P+4P = 8P = 2^3P$, $\ldots$, $2^{255}P$ (255 additions)
  - Write N as binary expansion, e.g.:
    - $N = 233 = 2^7 + 2^6 + 2^5 + 2^3 + 2^0$
    - $NP = 2^7P + 2^6P + 2^5P + 2^3P + 2^0P$
    - In this example, there are 4 point additions
    - Maximum number of point additions for 256-bit N is 255
  - Calculate NP using the binary expansion
  - Maximum number of point additions for 256-bit N: 255 + 255 = 510
- If N is 256-bit random number:
- Attacker: $\approx 2^{256}$ point additions (practically impossible)

- User: $\approx 2 \times 256$ point additions (easy)

In summary, knowing the b-bit value N, the user needs to perform about $2 \times b$ point additions. This is easy. But the attacker, who doesn't know N, must perform about 2^b point additions, which is practically impossible.

- The above discussed a normal elliptic curve
- But to ensure all values contained within finite coordinate space, modular arithmetic is used
- $y^2 \bmod p = (x^3 + ax + b) \bmod p$
- p is a prime number

The figures and examples given previously shown an elliptic curve without modular arithmetic. But in elliptic curve cryptography, modular arithmetic occurs. The same principles, and reasoning why it is hard for the attacker, still apply. The plots of the elliptic curve in modular arithmetic look different however—they now have distinct points in a finite coordinate space. Search online for examples.

Next we will see how elliptic curves are applied to build cryptographic mechanisms.

15.2 Applications of Elliptic Curve Cryptography

ECC can be applied for various cryptographic mechanisms:

- Secret key exchange, e.g. ECDH, ECMQV
- Digital signatures, e.g. ECDSA, EC-KCDSA
- Public key encryption, e.g. ECIES, PSEC

The most common applications are for secret key exchange, especially with Elliptic Curve Diffie-Hellman (ECDH), and digital signatures with Elliptic Curve Digital Signature Algorithm (ECDSA). We will look at ECDH in the following.

Algorithm 15.1 (Elliptic Curve Diffie-Hellman Key Exchange)**.** Assume users A and B have EC key pairs: $PU_A = NP$, $PR_A = N$, $PU_B = MP$, $PR_B = M$.

1. User A calculates secret $S_A = N \cdot PU_B = NMP$ using shortcut point addition.
2. User B calculates secret $S_B = M \cdot PU_A = MNP$ using shortcut point addition.

Diffie-Hellman key exchange can be used using ECC so that both users obtain a shared secret over an insecure channel. Users agree on a public point P. They generate their own keypairs, where the private key is some large random number, and the public key is that number times P. Note that in the key generation, each user can use the shortcut to calculate NP or MP.

Assume the users exchange public keys. They then use their own private key multiplied by the other's public key. Again, the shortcut point addition can be used. Both will arrive at the same point (coordinate), i.e. $NMP = MNP$. This is the shared secret.

An attacker that knows the public keys and initial point P has to find either N or M. If those numbers are large enough, this is practically impossible.

Until now we have referred to general or example elliptic curves without specifying the parameter values. In practice, users of ECC do not select their own parameters, but rather use standardised parameters.

- Parameters for ECC are usually standardised
  - Base point, P (also referred to as generator, G)
  - Curve parameters, a and b
  - Prime, p
  - Other parameters also included
- Common curves (see also `https://safecurves.cr.yp.to/`):
  - NIST FIPS 186: P-256, P-384 and 13 others
  - SECG: secp160k1, secp160r1, . . . (NIST curves are a subset)
  - ANSI X9.62: prime192, prime256, . . .
  - Other curves: Curve25519, Brainpool

SECG in SEC 2 defined a large set of curves. The NIST curves were a subset of the SEC 2 curves. NSA Suite B curves are a subset of NIST curves.

15.3 Elliptic Curve Cryptography in OpenSSL

In OpenSSL, the `ecparam` command is used for elliptic curve cryptography parameter and key generation. Many of the operations are explained the OpenSSL Command Line Elliptic Curve Operations wiki. You can list the curves supported by OpenSSL:

```
alice@node1:~$ openssl ecparam -list_curves
  secp112r1 : SECG/WTLS curve over a 112 bit prime field
  secp112r2 : SECG curve over a 112 bit prime field
  secp128r1 : SECG curve over a 128 bit prime field
  secp128r2 : SECG curve over a 128 bit prime field
  ...
  brainpoolP512r1: RFC 5639 curve over a 512 bit prime field
  brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
  SM2       : SM2 curve over a 256 bit prime field
```

For a selected curve, you can see the detailed parameters. For example, for the secp256k1 curve:

```
alice@node1:~$ openssl ecparam -name secp256k1 -text -param_ecn explicit -noout
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
```

```
B:    7 (0x7)
Generator (uncompressed):
    04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
    0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
    8f:fb:10:d4:b8
Order:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor: 1 (0x1)
```

Public/private key pairs can be generated from named curve (e.g. secp256k1) or by first outputting curve parameters to a file. Here we will show the latter:

```
alice@node1:~$ openssl ecparam -name secp256k1 -out secp256k1.pem
alice@node1:~$ openssl ecparam -in secp256k1.pem -genkey -noout -out alice-k
ey.pem
```

Alternatively, you could combine the above two commands into a single, by specifying the `-name` of the curve rather than the `-in` file. The OpenSSL Command Line Elliptic Curve Operations wiki explains the different options, as well as ensuring parameters are in a format that can be used by different versions of OpenSSL.

Once the curve parameters file (e.g. `secp256k1.pem` is generated, you can use the `genpkey`, `key` and `pkeyutl` operations in a similar manner as with Diffie-Hellman in Section 14.5.

Part V

Authentication

Chapter 16

Hash Functions and MACs

This chapter introduces two primitives used in authentication and data integrity: cryptographic hash functions and Message Authentication Codes. While these primitives can be based on symmetric key ciphers (and occasionally public key ciphers), in many cases they are custom-designed algorithms to meet the specific needs for authentication.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

16.1 Informal Overview of Hashes and MACs

We will start with an informal overview of the concepts, terminology, security goals and applications. We will primarily refer to *hash functions* and *Message Authentication Codes*.

- Hash functions
  - Takes message as input and returns short, unique and random-looking output
  - Different inputs will produce different outputs
  - Also called: Modification Detection Code (MDC), unkeyed hash function
  - Output called: hash (h), digital fingerprint, imprint, message digest
  - $h = H(M)$
- Message Authentication Code (MAC)
  - Takes message *and a secret key* as input and returns short, unique and random-looking output
  - Different inputs (key and/or data) will produce different outputs
  - Also called: keyed hash function
  - Output called: tag (t), code or MAC
  - $t = MAC(K, M)$

File: crypto/hash.tex, r1951

Chapter 9 of the Handbook of Applied Cryptography explains the different classifications of hash functions.

Also note that our focus is on cryptographic purposes of hashes and MACs. They have other, non-crypto applications, e.g. hash functions for caching. To be more precise we should refer to *cryptographic hash functions*, however for brevity we often just refer to *hash functions*.

- Pre-image resistance (one-way)
  - Given the output (hash/tag), attacker cannot find the input message
- Second pre-image resistance (weak collision resistance)
  - Given one message, attacker cannot find another message with same output (hash/tag)
- Collision resistance (strong collision resistance)
  - Attacker cannot find any two messages that produce same output (hash/tag)

Note that there is different terminology used for the properties. The names in parentheses are an alternative form.

The first two properties are similar from a security perspective: most algorithms that have one property also have the other. However the third property of (strong) collision resistance is harder to provide. That is, some algorithms may have the first two properties, but not the third of (strong) collision resistance.

- Digital signature (public key crypto + hash)
  - preimage, 2nd preimage, collision resistance (if attacker can perform chosen message attack)
- Message authentication with symmetric key encryption and hash
  - none
- Message authentication with MAC only
  - preimage, 2nd preimage, collision resistance (if attacker can perform chosen message attack)
- Message authentication using hash only
  - Assumes an authentic channel, where delivery of hash is trusted
  - 2nd preimage resistant
- Password storage with hash
  - preimage resistant

16.2 Introduction to Hash Functions

Hash functions are algorithms used in different aspects of computing and IT, and especially important in cryptography. We often distinguish between different hash functions used for general computing purposes versus those used in cryptography based on the properties of the function.

- Hash function or algorithm H():
  - Input: variable-length block of data M
  - Output: fixed-length, small, hash value, h, where $h = \mathrm{H}(M)$
  - Another name for hash value is digest
  - Output hash values should be evenly distributed and appear random
- A secure, cryptographic hash function is practically impossible to:
  - Find the original input given the hash value
  - Find two inputs that produce the same hash value

A hash function is an algorithm that usually takes any sized input, like a file or a message, and produces a short (e.g. 128 bit, 512 bit) random looking output, the hash value. If you apply the hash function on the same input, you will always get the exact same hash value as output. In practice, if you apply the hash function on two different inputs, you will get two different hash values as output.

- Message authentication
- Digital signatures
- Storing passwords
- Signatures of data for malicious behaviour detection (e.g. virus, intrusion)
- Generating pseudorandom number

Hash functions are important in many areas of security. They are typically used to create a fingerprint/signature/digest of some input data, and then later that fingerprint is used to identify if the data has been changed. However they also have uses for hiding original data (storing passwords) and generating random data. Different applications may have slightly different requirements regarding the security (and performance) properties of hash functions.

There are three general approaches to design hash functions:

Based on Block Ciphers Well-known and studied block ciphers are used with a mode of operation to produce a hash function. Generally, less efficient than customised hash functions.

Based on Modular Arithmetic Similar motivation as to basing on block ciphers, but based on public key principles. Output length can be any value. Precautions are needed to prevent attacks that exploit mathematical structure.

Customised Hash Functions Functions designed for the specific purpose of hashing. Disadvantage is they haven't been studied as much as block ciphers, so harder to design secure functions.

Designing hash functions based on existing cryptographic primitives is advantageous in that existing knowledge and implementations can be re-used. However as more time has been spent studying customised hash functions, they are now the approach of choice due to their security and efficiency.

While a number of cryptographic hash functions have been designed over the years, the two main ones of interest are MD5 and SHA.

	Output	Classification	
Primitive	Length	Legacy	Future
SHA-2	256, 384, 512, 512/256	✓	✓
SHA-3	256, 384, 512	✓	✓
SHA-3	SHAKE128, SHAKE256	✓	✓
Whirlpool	512	✓	✓
BLAKE	256, 384, 512	✓	✓
RIPEMD-160	160	✓	✗
SHA-2	224, 512/224	✓	✗
SHA-3	224	✓	✗
MD5	128	✗	✗
RIPEMD-128	128	✗	✗
SHA-1	160	✗	✗

Credit: ECRYPT CSA Algorithms, Key Size and Protocols Report, 2018

Figure 16.1: Selected Cryptographic Hash Functions

Figure 16.1 shows selected hash functions, classified for legacy or future use. It is taken from the ECRYPT-CSA 2018 report on Algorithms, Key Sizes and Protocols. The authors classified hash functions as legacy, meaning secure for near future, and future, meaning secure for medium term. It includes history hash functions no longer recommended, such as MD5, RIPEMD-128 and SHA-1. There are many other hash functions. Wikipedia has a nice comparison.

16.3 Properties of Cryptographic Hash Functions

Definition 16.1 (Pre-image of a Hash Value)**.** For hash value $h = \mathrm{H}(x)$, x is pre-image of h. As H is a many-to-one mapping, h has multiple pre-images. If H takes a b-bit input, and produces a n-bit hash value where $b > n$, then each hash value has 2^{b-n} pre-images.

A hash function takes a single input and produces a single output. The output is the hash value and the input is the pre-image of that hash value.

Definition 16.2 (Hash Collision)**.** A collision occurs if $x \neq y$ and $\mathrm{H}(x) = \mathrm{H}(y)$. Collisions are undesirable in cryptographic hash functions.

We will show shortly that collisions should be practically impossible to be found by an attacker.

Exercise 16.1 (Number of Collisions)**.** If H_1 takes fixed length 200-bit messages as input, and produces a 80-bit hash value as output, are collisions possible?

Solution 16.1 (Number of Collisions)**.** Yes. In this simplistic example (since hash functions normally take variable length messages), there are 2^{200} possible different inputs. A hash function maps an input to an output hash value. There are 2^{80} possible output hash values. That means at least two of the inputs must map to the same output hash value, i.e. a collision. Assuming the hash function distributes the pre-images to hash values in a uniformly random manner, then on average, each hash value has $2^{200-80} = 2^{120}$ pre-images.

The point is, that if the input message length is larger than the output hash value (and in practice, it always is), then collisions are theoretically possible. One aspect of designing cryptographically secure hash functions is to make it practical impossible for an attacker to find useful collisions.

Now let's restate the general requirements of a cryptographic hash function.

Variable input size: H can be applied to input block of any size

Fixed output size: H produces fixed length output

Efficiency: $\mathrm{H}(x)$ relatively easy to compute (practical implementations)

Pseudo-randomness: Output of H meets standard tests for pseudo-randomness

Properties: Satisfies one or more of the properties: Pre-image Resistant, Second Pre-image Resistant, Collision Resistant

Now let's define several common required properties of cryptographic hash functions.

Definition 16.3 (Pre-image Resistant Property)**.** For any given h, it is computationally infeasible to find y such that $\mathrm{H}(y) = h$. Also called the *one-way property.*

Informally, it is hard to inverse the hash function. That is, given the output hash value, find the original input message.

Definition 16.4 (Second Pre-image Resistant Property)**.** For any given x, it is computationally infeasible to find $y \neq x$ with $\mathrm{H}(y) = \mathrm{H}(x)$. Also called *weak collision resistant* property.

To break this property, the attacker is trying to find a collision. That is, two input messages x and y that produce the same output hash value. Importantly, the attacker cannot choose x. They are given x and must find a different message y that produces a collision.

Definition 16.5 (Collision Resistant Property)**.** It is computationally infeasible to find any pair (x, y) such that $\mathrm{H}(x) = \mathrm{H}(y)$. Also called *strong collision resistant* property.

To break this property, again the attacker is trying to find a collision. However in this case the attacker has the freedom to find *any* messages x and y that produce a collision. This freedom makes it easier for the attacker to perform an attack against this property than against the Second Pre-image Resistant property.

- Pre-image and Second Pre-image Attack
  - Find a y that gives specific h; try all possible values of y
  - With b-bit hash code, effort required proportional to 2^b
- Collision Resistant Attack
  - Find any two messages that have same hash values
  - Effort required is proportional to $2^{b/2}$
  - Due to birthday paradox, easier than pre-image attacks

Exercise 16.2 (Brute Force Attack on Hash Function)**.** Consider a hash function to be selected for use for digital signatures. Assume an attacker has compute capabilities to calculate 10^{12} hashes per second and is prepared to wait for approximately 10 days for a brute attack. Find the minimum hash value length that the hash function should support, such that a brute force is not possible.

Solution 16.2 (Brute Force Attack on Hash Function)**.** There are two cases to consider. If the hash function and network is subject to a chosen message attack, then the hash function should support all three properties. Preimage and Second Preimage Resistant properties required effort of approximately 2^b for the attacker. But attacking Collision Resistant property requires significantly less effort, $2^{b/2}$. Therefore the hash length, b, must be sufficient so that an attack on Collision Resistant property is not possible.

$$\begin{aligned}
\frac{2^{b/2}}{10^{12}} &> 10 \times 24 \times 60 \times 60 \\
2^{b/2} &> 10^{12} \times 10 \times 86400 \\
b/2 &> log_2(8.64 \times 10^{17}) \\
b/2 &> 59.583 \\
b &> 119.168
\end{aligned}$$

Therefore the hash length, b, should be at least 120 bits.

If however a chosen message attack is not possible, then the hash function only needs to meet the Preimage and Second Preimage Resistant properties. Therefore only a 60 bit hash length is needed (i.e. a weaker hash function).

16.4 Introduction to Message Authentication Codes

In the above we looked at cryptographic hash functions that take a message as input and produce a hash value as output. One application of hash functions is authentication, as we will see in depth in Chapter 17. However note that hash functions do not use any secret key as input. A variation is to introduce a secret key as input, resulting in a keyed hash function.

- Hash functions have no secret key
  - Can be referred to as unkeyed hash function
  - Also called Modification Detection Code
- A variation is to allow a secret key as input, in addition to the message
  - $h = \mathrm{H}(K, M)$
  - Keyed hash function or Message Authentication Code (MAC)
- Hashes and MACs can be used for message authentication, but hashes also used for multiple other purposes
- MACs are more common for authentication messages

Now we will shift our focus to MACs, first looking at the general design approaches.

Based on Block Ciphers CBC-MAC, OMAC, PMAC,

Customised MACs MAA, MD5-MAC, UMAC, Poly1305

Based on Hash Functions HMAC

The motivation for different design approaches is similar to that for hash function design approaches.

Definition 16.6 (Computation Resistance of MAC)**.** Given one or more text-tag pairs, $[x_i, \mathrm{MAC}(K, x_i)]$, computationally infeasible to compute any text-tag pair $[y, \mathrm{MAC}(K, y)]$, for a new input $y \neq x_i$

Assume an attacker has intercepted messages (text) and the corresponding MACs (tags). They have i such text-tag pairs. Now there is a new message y. It should be practically impossible for the attacker to find the corresponding tag of y, that is, $\mathrm{MAC}(K, y)$.

Given what the attacker must do, the security of MACs can be defined based on the effort of brute force attacks.

- Brute Force Attack on Key
  - Attacker knows $[x_1, T_1]$ where $T_1 = MAC(K, x_1)$
  - Key size of k bits: brute force on key, 2^k
  - But ... many tags match T_1
  - For keys that produce tag T_1, try again with $[x_2, T_2]$
  - Effort to find K is approximately 2^k

- Brute Force Attack on MAC value
  - For x_m, find T_m without knowing K
  - Similar effort required as one-way/weak collision resistant property for hash functions
  - For n bit MAC value length, effort is 2^n
- Effort to break MAC: $\min(2^k, 2^n)$

Chapter 17

Authentication and Data Integrity

This chapter shows how messages can be authenticated, including ensuring data integrity, using various cryptographic primitives, especially hash functions and MACs from Chapter 16.

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

17.1 Aims of Authentication

There are different types of attacks that can occur with information transfer. In turn, different mechanisms are available to prevent/detect such attacks.

1. Disclosure: encryption
2. Traffic analysis: encryption
3. Masquerade: message authentication
4. Content modification: message authentication
5. Sequence modification: message authentication
6. Timing modification: message authentication
7. Source repudiation: digital signatures
8. Destination repudiation: digital signatures

We have cover encryption primarily from the perspective of preventing disclosure attacks, i.e. providing confidentiality. Now we will look at preventing/detecting masquerade, modification and repudiation attacks using authentication techniques. Note that we consider digital signatures as a form of authentication.

- Receiver wants to verify:
    1. Contents of the message have not been modified (*data authentication*)

File: crypto/auth.tex, r1951

2. Source of message is who they claim to be (*source authentication*)

- Different approaches available:
  - Symmetric Key Encryption
  - Hash Functions
  - Message Authentication Codes (MACs)
  - Public Key Encryption (i.e. Digital Signatures)

We will cover these different approaches in the following sections.

17.2 Authentication with Symmetric Key Encryption

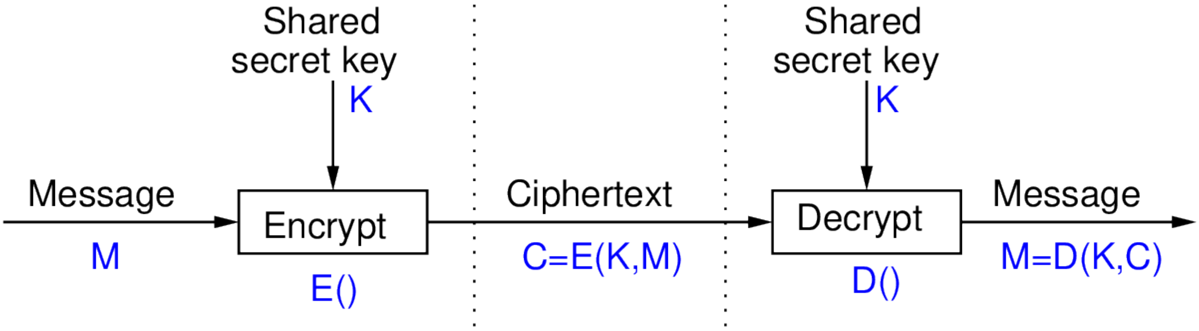


Figure 17.1: Symmetric Encryption for Authentication

Figure 17.1 shows symmetric key encryption used for confidentiality. On the left is the sender A, and on the right is the receiver B. In the middle (between the dashed lines) is the information sent from A to B. Only B (and A) can recover the plaintext. However in some cases this also provides:

- Source Authentication: A is only other user with key; B knows it must have come from A
- Data Authentication: successfully decrypted implies data has not been modified

The source and data authentication assumes that the decryptor (B) can recognise that the result of the decryption, i.e. the output plaintext, is correct.

The assumption about being able to recognise the correct plaintext is explored next.

Question 17.1 (Recognising Correct Plaintext in English)**.** *B* receives ciphertext (supposedly from *A*, using shared secret key *K*):

`DPNFCTEJLYONCJAEZRCLASJTDQFY`

B decrypts with key *K* to obtain plaintext:

`SECURITYANDCRYPTOGRAPHYISFUN`

Was the plaintext encrypted with key *K* (and hence sent by *A*)? Is the ciphertext received the same as the ciphertext sent by *A*?

The typical answer for above is yes, the plaintext was sent by A and nothing has been modified. This is because the plaintext "makes sense". Our knowledge of most ciphers (using the English language) is that if the wrong key is used or the ciphertext has been modified, then decrypting will produce an output that does not make sense (not a combination of English words).

Question 17.2 (Recognising Correct Plaintext in English). B receives ciphertext (supposedly from A, using shared secret key K):

`QEFPFPQEBTOLKDJBPPXDBPLOOVX`

B decrypts with key K to obtain plaintext:

`FTUEUEFTQIDAZSYQEEMSQEADDKM`

Was the plaintext encrypted with key K (and hence sent by A)? Is the ciphertext received the same as the ciphertext sent by A?

Based on the previous argument, the answer is no. Or more precise, either the plaintext was not sent by A, or the ciphertext was modified along the way. This is because the plaintext makes no sense, and we were expected it to do so.

Question 17.3 (Recognising Correct Plaintext in Binary). B receives ciphertext (supposedly from A, using shared secret key K):

`0110100110101101010110111000010`

B decrypts with key K to obtain plaintext:

`0101110100001101001010100101110`

Was the plaintext encrypted with key K (and hence sent by A)? Is the ciphertext received the same as the ciphertext sent by A?

This is harder. We cannot make a decision without further understanding of the expected structure of the plaintext. What are the plaintext bits supposed to represent? A field in a packet header? A portion of a binary file? A random key? Without further information, the receiver does not know if the plaintext is correct or not. And therefore does not know if the ciphertext was sent by A and has not been modified.

- Many forms of information as plaintext can be recognised at correct
- However not all, and often not automatically
- Authentication should be possible without decryptor having to know context of the information being transferred
- Authentication purely via symmetric key encryption is insufficient
- Solutions:
  - Add structure to information, such as error detecting code
  - Use other forms of authentication, e.g. MAC

We will see some of the alternatives in the following sections.

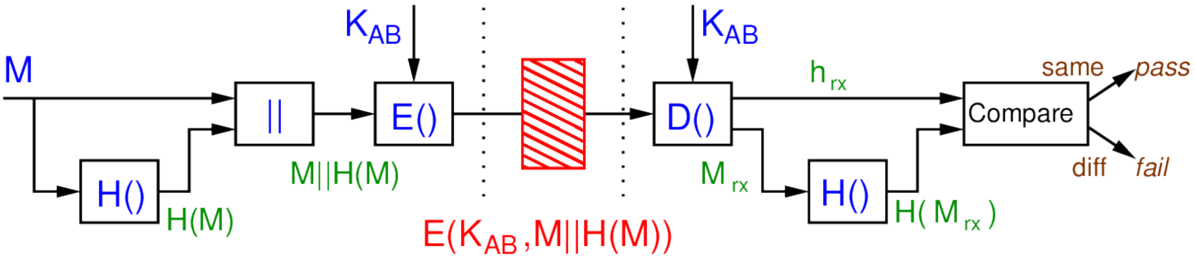


Figure 17.2: Authentication by Hash and then Encrypt

17.3 Authentication with Hash Functions

Figure 17.2 shows a scheme where the hash function is used to add structure to the message. Again, user A and B are on the left and right, respectively. The inputs (message and secret key) and operations are shown in blue. The green values are used to refer to intermediate values. In the middle in red is the information sent from A to B.

At the receiver, the "received" message and hash are denoted with a subscript *rx*. In the normal case (no attack or error), the received values will be identical to the sent values, i.e. $M_{rx} = M$. However if an attack takes place, then it is possible the sent and received values differ.

When the receiver decrypts, they will be able to determine if the plaintext is correct by comparing the hash of the message component with the stored hash value. This is one method of addressing the problem of using just symmetric key encryption on its own for authentication. This scheme provides confidentiality of the message and authentication.

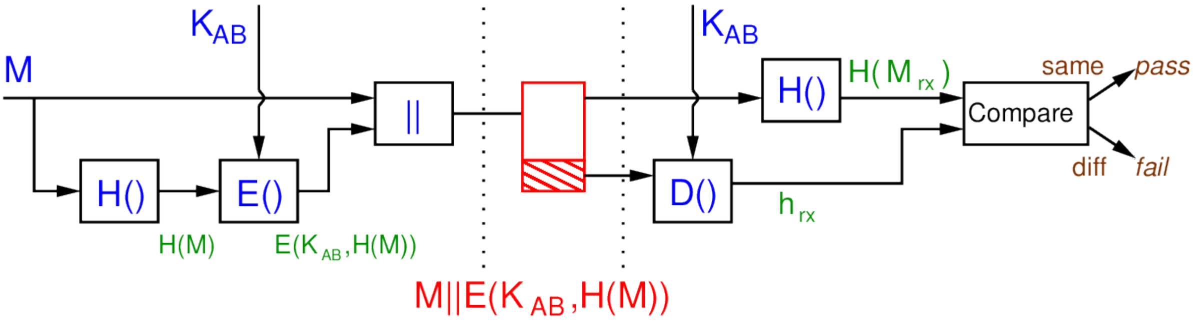


Figure 17.3: Authentication by Encrypting a Hash

Figure 17.3 shows a different scheme where only the hash value is encrypted. The receiver can verify that nothing has been changed. This scheme provides authentication, but does not attempt to provide confidentiality. This is useful in reducing any computation overhead when confidentiality is not required.

Exercise 17.1 (Attack of Authentication by Encrypting a Hash)**.** If a hash function did not have the Second Preimage Resistant property, then demonstrate an attack on the scheme in Figure 17.3.

Solution 17.1 (Attack of Authentication by Encrypting a Hash)**.** The attacker intercepts the message $M||\mathrm{E}(K, \mathrm{H}(M))$ before it reaches B. If the Second Preimage Resistant property does not hold, then it is possible for an attacker to find another message M'

where H(M) = H(M'). As a result, the attacker can modify M to M', but leave the remainder of the sent information, E(K, H(M)) as is. They forward M'||E(K, H(M)) to B. User B decrypts with the key shared with A, then compare the hash value with H(M'). They match. Therefore user B trusts the message, but in fact it has been subject to a modification attack.

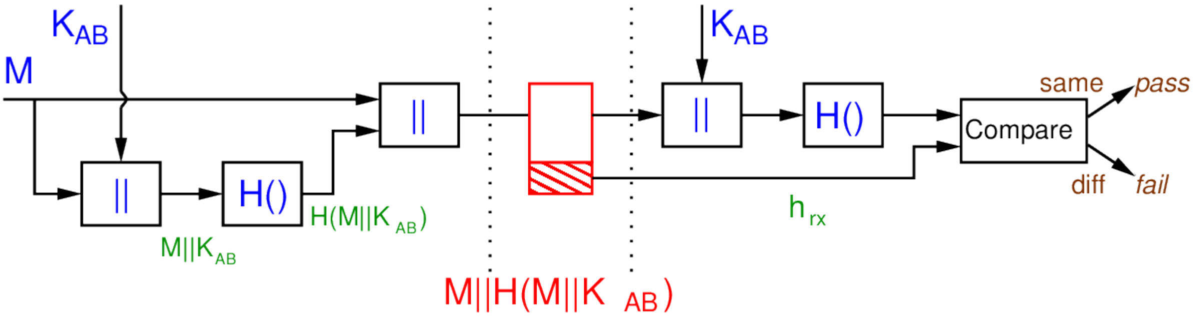


Figure 17.4: Authentication with Hash of a Shared Secret

Figure 17.4 shows a scheme the provides authentication, but without using any encryption. Avoiding encryption can be desirable in very resource constrained environments. S is a secret value shared by A and B. Concatenating the secret with the message, and then hashing the result, allows the receiver the verify the plaintext is correct, and keeps the secret confidential.

Exercise 17.2 (Attack of Authentication with Hash of Shared Secret)**.** If a hash function did not have the Preimage Resistant property, then demonstrate an attack on the scheme in Figure 17.4.

Solution 17.2 (Attack of Authentication with Hash of Shared Secret)**.** The attacker intercepts the message M||H(M||S). If the Preimage Resistant property does not hold, then it is possible for an attacker, given a hash value, to find the original input, i.e. the preimage. That is the attacker find $M||S$. Since they also know M, it is easy to find S, i.e. the remaining bits. The attacker now knows the shared secret and could masquerade as A.

In Section 17.5 we will see the role of hash functions in digital signatures.

17.4 Authentication with MACs

MACs can be used for authentication by themselves, or combined with symmetric key encryption (e.g. when confidentiality is also required). First we look at using only MACs.

Figure 17.5 shows a scheme where authentication is provided using only a MAC. That is, encryption is not used.

Now we consider the case of combining MACs with encryption.

- Common to what both confidentiality and authentication (data integrity)
- MACs have advantage over hashes in that if encryption is defeated, then MAC still provides integrity

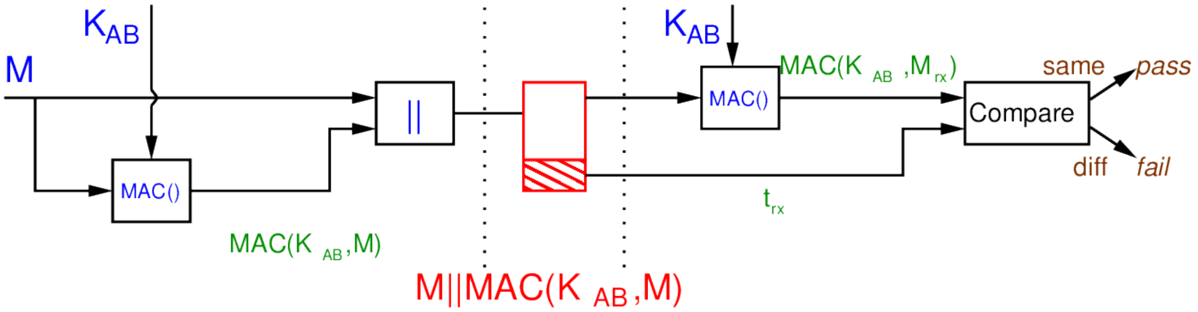


Figure 17.5: Authentication with only MACs

- But two keys must be managed: encryption key and MAC key
- Recommended algorithms used for encryption and MAC are independent
- Three general approaches (following definitions), referred to as authenticated encryption

Definition 17.1 (Encrypt-then-MAC)**.** The sender encrypts the message M with symmetric key encryption, then applies a MAC function on the ciphertext. The ciphertext and the tag are sent, as follows:

$$\mathrm{E}(K_1, M)||\mathrm{MAC}(K_2, \mathrm{E}(K_1, M))$$

Two independent keys, K_1 and K_2, are used.

Definition 17.2 (MAC-then-Encrypt)**.** The sender applies a MAC function on the plaintext, appends the result to the plaintext, and then encrypt both. The ciphertext is sent, as follows:

$$\mathrm{E}(K_1, M||\mathrm{MAC}(K_2, M))$$

Definition 17.3 (Encrypt-and-MAC)**.** The sender encrypts the plaintext, as well ass applying a MAC function on the plaintext, then combines the two results. The ciphertext joined with tag are sent, as follows:

$$\mathrm{E}(K_1, M)||\mathrm{MAC}(K_2, M)$$

Which of the three approaches is better?

- There are small but important trade-offs between encrypt-then-MAC, MAC-then-encrypt and encrypt-and-MAC
- Potential attacks on each, especially if a mistake in applying them
- Generally, encrypt-then-MAC is recommended, but are cases against it
- Some discussion of issues:
  - Chapter 9.6.5 of Handbook of Cryptography
  - Moxie Marlinspike

  - StackExchange
  - Section 1 and 2 of Authenticated Encryption by J Black

- Other authenticated encryption approaches incorporate authenticate into encryption algorithm
  - AES-GCM, AES-CCM, ChaCha20 and Poly1305

It is worth reading some of the discussion about the three approaches.

17.5 Digital Signatures

- Authentication has two aims:
  - Authenticate data: ensure data is not modified
  - Authenticate users: ensure data came from correct user
- Symmetric key crypto, MAC functions are used for authentication
  - But cannot prove which user created the data since two users have the same key
- Public key crypto for authentication
  - Can prove that data came from only 1 possible user, since only 1 user has the private key
- Digital signature
  - *Encrypt hash of message using private key of signer*

A digital signature has the same purpose of a handwritten signature: to prove that a document (or message or file) is approved by and originated from one particular person. If a message is signed, the signer cannot claim they did not sign it (since they are the only person that could create the signature). Similar, someone cannot pretend to be someone else, since they cannot create that other persons signature. Of course handwritten signatures are imprecise and sometimes forgeable. Digital signatures are much more secure, making it practically impossible for someone to forge a signature or modify a signed document without it being noticed.

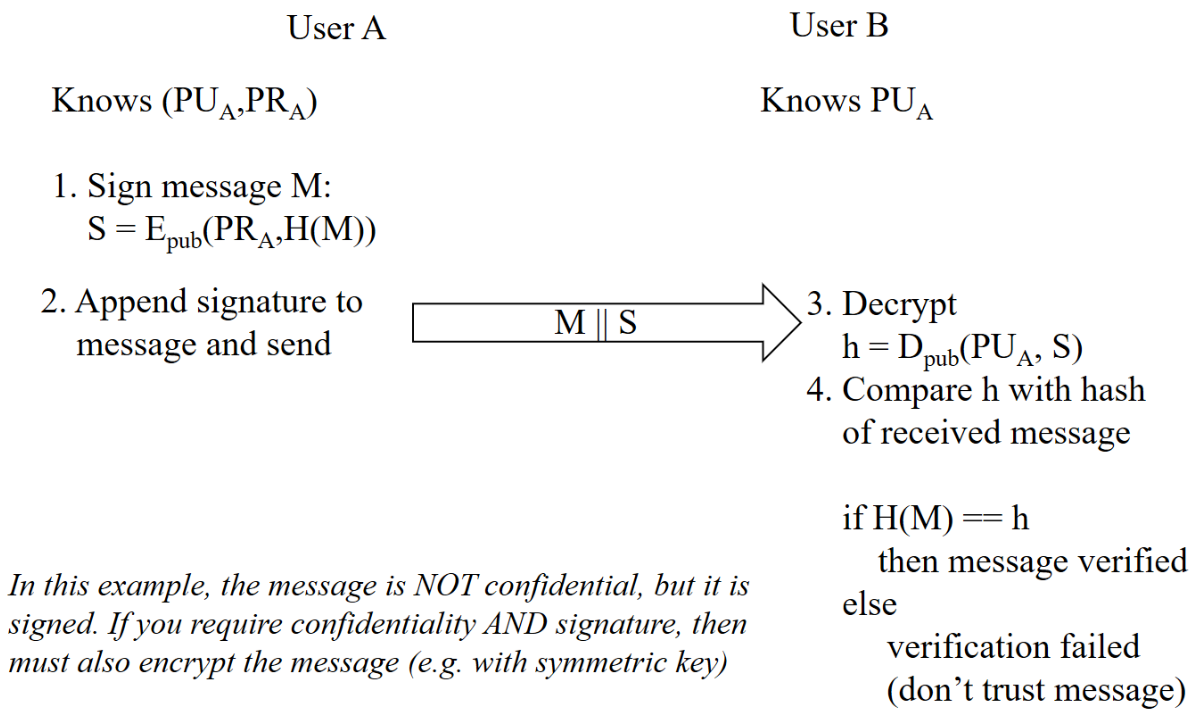
In practice, a digital signature of a message is created by first calculating a hash of that message, and then encrypting that hash value with the private key of the signer. The signature is then attached to the message.

The hash function is not necessary for security, but makes signatures practical (the signature is short fixed size, no matter how long the message is).

- User A has own key pair: (PU_A, PR_A)
- Signing

  - User A signs a message by encrypting hash of message with own private key: $S = E(PR_A, H(M))$
  - User attaches signature S to message M and sends to user B

- Verification
  - User B verifies a message by decrypting signature with signer's public key: $h = D(PU_A, S)$
  - User B then compares hash of received message, $H(M)$, with decrypted h; if identical, signature is verified



In this example, the message is NOT confidential, but it is signed. If you require confidentiality AND signature, then must also encrypt the message (e.g. with symmetric key)

Figure 17.6: Digital Signature Example

Part VI

Key Management

Chapter 18

Key Distribution and Management

To be completed in the future.

18.1 Recommended Key Sizes

Users of cryptographic systems need to know what algorithms and parameters that should use, without having to understand the details.

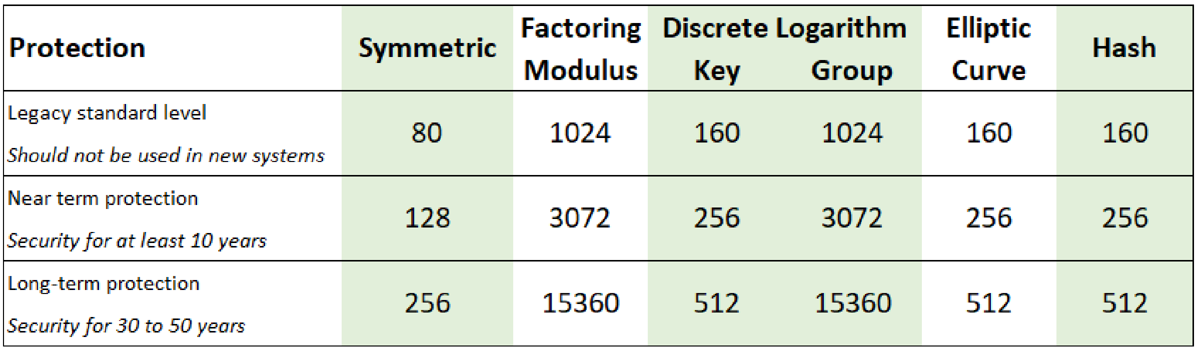
- Various governments, standardisation organisations and researchers have analysed security level of cryptographic mechanisms
- Provide recommendations for:
  - Ciphers to use
  - Key lengths or hash lengths
  - Security level
- BlueKrypt website summarises recommendations: www.keylength.com
  - E.g. from NIST, German BSI, NSA, ECRYPT project, ...
- ECRYPT-CSA Project 2018 report on Algorithms, Key Size and Protocols (PDF)

The BlueKrypt website summarises recommendations from various organisations. You should visit the website and explore the different recommendations. While there are differences, you can get an approximate idea of the key lengths that should be used.

The ECRYPT-CSA project is one effort to compare algorithms. The PDF report gives a comprehensive summary of different cryptographic mechanisms, analysis of specific algorithms, and recommendations.

Figure 18.1 shows recommended key (or hash) lengths, in bits, for symmetric key algorithms (e.g. AES), public key algorithms based on factoring a modulus (e.g. RSA), public key algorithms based on solving discrete logarithms (e.g. the secret key and modulus/group length in Diffie-Hellman), public key algorithms based on elliptic curve cryptography, and hash functions.

File: crypto/keys.tex, r1969



Protection	Symmetric	Factoring	Discrete Logarithm		Elliptic	Hash
		Modulus	Key	Group	Curve	
Legacy standard level *Should not be used in new systems*	80	1024	160	1024	160	160
Near term protection *Security for at least 10 years*	128	3072	256	3072	256	256
Long-term protection *Security for 30 to 50 years*	256	15360	512	15360	512	512

Credit: BlueKrypt www.keylength.com, CC-BY-SA 3.0

Figure 18.1: Recommend Key Lengths from ECRYPT-CSA 2018

Three different levels of security are given: legacy, current (near-term) and future (long-term). Current or future levels of security should be used, although legacy levels may still be secure for some cases.

Chapter 19

Digital Certificates

To be added in the future.

File: crypto/cert.tex, r1792

Part VII

Advances in Cryptography

Chapter 20

Quantum Computing and Cryptography

This chapter contains brief notes on concepts related to quantum computing and quantum cryptography. The intention is to be able to understand the role of quantum computing with respect to attacking ciphers, as well as the security mechanism quantum cryptography provides.

Disclaimer: These are very rough notes. A lack of time and in-depth understanding of quantum computing on my part means there are likely errors, some parts may be confusing, and the presentation is quite poor (mainly definitions which are not actually definitions; insufficient examples or diagrams). However it should be enough to gain an idea what role quantum technology plays in cryptography. The plan is to update the content after feedback from others.

This information comes from a collection of resources, including Wikipedia pages, news articles and videos. Some, but definitely not all, of those sources are, in no particular order:

- https://quantum.country/
- https://blogs.iu.edu/sciu/2019/07/13/quantum-computing-parallelism/
- https://arxiv.org/abs/quant-ph/0507023
- http://www.columbia.edu/~jpp2139/decoherence-superconducting-qubitsWEBv2.pdf
- https://www.ibm.com/quantum-computing/
- https://qiskit.org/textbook/preface.html

Presentation slides that accompany this chapter can be downloaded in the following formats: slides only (PDF); slides with notes (PDF, ODP, PPTX).

File: crypto/quantum.tex, r1971

20.1 Quantum Computing

Definition 20.1 (Quantum Technology)**.** Emerging technologies that build upon concepts of quantum physics, especially superposition and entanglement. Includes quantum computing and quantum cryptography.

Note that before quantum physics we had "classical" physics. Similar, we will differentiate between quantum computers and classical computers (those that we know and use everyday). Also, roughly, quantum physics and quantum mechanics means the same thing in this discussion, and we refer to quantum-mechanical systems.

To arrive at an explanation of a quantum computer, as well as quantum cryptography, we will step through some of the basic principles/ideas. First we will look at how information is represented in

Definition 20.2 (bit)**.** Binary digit, 0 or 1, as the basic unit of information in classical computers. For example stored as electric charges in capacities or with magnets in hard disks. Communicated with electrical or optical pulses. A bit has two states: 0 or 1.

A bit is defined, in an informal manner, just for reference.

Definition 20.3 (qubit)**.** Quantum bit has states represented in a quantum-mechanical system. The state of a qubit is a vector. A qubit has two *basis states*, $|0\rangle$ and $|1\rangle$, but many possible states in between. Often represented using subatomic particles such as electrons or photons.

The key distinguishing feature of qubits compared to bits is that qubits have many possible states, not just 0 and 1.

The notation used is not so important here; it is just a short way that we can identify the two basis states which are similar to bit 0 and bit 1. We will see next how the qubit is expressed when in the "in between" states.

Definition 20.4 (Quantum Superposition)**.** Any two (or more) quantum states can be added together to form another quantum state. That result is a superposition of the original states.

Superposition is a concept seen in other systems, but quantum superposition is the main concept that delivers powerful innovations with quantum computers.

Example 20.1 (qubit Superposition)**.** Basis state $|0\rangle$ is like bit 0. Basis state $|1\rangle$ is like bit 1. The state $0.6|0\rangle + 0.8|1\rangle$ is an example of a superposition of the two basis states, and forms another state of the qubit. Another example state is $0.866|0\rangle + 0.5|1\rangle$. In general, a superposition state is $\alpha|0\rangle + \beta|1\rangle$, where $\alpha^2 + \beta^2 = 1$.

You may think of the concept as superposition as follows. A classical bit has the value 0 or 1. A qubit has the value of 0 or 1, or a value that is both 0 and 1 at the same time.

An important point is that the weights, α and β, can be controlled. This is the key part of how qubits are used in calculations, as next we see that measuring a qubit returns 0 or 1 with some probability.

Definition 20.5 (The Measurement Problem)**.** Measuring a qubit gives the bit 0 with probability α^2 and bit 1 with probability β^2. After measurement the qubit enters (collapses into) the basis state.

There are two important issues about measuring a qubit. First, the result will either be 0 or 1. However when the qubit is in a superposition state of $\alpha|0\rangle + \beta|1\rangle$, then we don't know in advance which value will be output from the measurement. But we do know that with probability α^2 it will be bit 0 and with probability β^2 it will be bit 1. By controlling the weights, α and β, we can increase the probability that a useful output will be measured.

The other issue is that upon measurement, the qubit reverts to one of the basis states. It will no longer be a superposition of states.

Definition 20.6 (Quantum Entanglement)**.** Pair of particles are dependent on each other, meaning the quantum state of one particle impacts on the other.

Quantum entanglement is another concept, which you may hear about when referring to quantum communications and quantum teleportation. We will not cover it in any depth here, but present a simple example in the following.

Entanglement can be achieved for example by firing a laser at a crystal that causes two photons to split but be entangled.

Example 20.2 (qubit Entanglement)**.** If 2 qubits are entangled, then if one qubit is measured to be 0, then the other qubit will also be measured to be 0 (and similar if measured as 1).

Experiments have had qubits entangled over distances of 10's of kilometres.

Now we have some properties of qubits, let's start to define how computations are performed.

Definition 20.7 (Quantum Computation (informal))**.** A quantum computation starts with a set of qubits, modifies their states to perform some intended calculation, and then measures the result.

This definition of quantum computation is quite vague. How are the states of the qubits modified? Using logic gates to form circuits. One point to note is that at the end the result is measured. As noted before, measuring a quantum system will return some binary value with some probability *and* collapes any superpositions. This means that any speed up to be potentially be obtained by quantum computing needs to be done before the measurement.

Definition 20.8 (Classical Computer Circuits)**.** Circuits in classical computers are built from logic gates, such as AND, NOT, OR, XOR, NAND and NOR.

Note that AND and NOT gates are the universal set: everything else can be built from them.

Definition 20.9 (Quantum Computer Circuits)**.** Circuits in quantum computers are built from quantum logic gates. Single-bit gates include NOT, Hadamard, Phase and Shift gates; two-bit gates include Controlled NOT and SWAP; as well as 3-qubit Toffoli and Fredkin gates. Not all quantum gates have analagous operation with classical gates.

A single-bit gate takes a single qubit as input and produces a single qubit as output.

Now we can arrive at a simple definition of a quantum computer.

Definition 20.10 (Quantum Computer)**.** A (digital) quantum computer is built from a set of quantum logic gates, i.e. quantum circuits, and is said to perform quantum computation on qubits. An analog quantum computer also operates on qubits, but rather than using logic gates, using concepts of quantum simulation and quantum annealing.

We are only covering a digital quantum computer. The topics of quantum simulation and quantum annealing are not covered here.

20.2 Quantum Algorithms

Definition 20.11 (Quantum Register)**.** A quantum register is a set of n qubits. With a classical 2-bit register, there are four possible states: 00, 01, 10 and 11. A quantum 2-bit register can be in all four states at one time, as it is a superposition of the four states: $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$. Measuring the register will return one of the four states, with probability depending on the weights.

For example, if the two qubits are constructed so that $\beta = 0$ and $\delta = 0$, and $\alpha = \gamma = 1/\sqrt{2}$, then there is 50% probability of measuring 00 and 50% probability of measuring 10. There is no chance of measuring 01 or 11.

Now we get to the benefit of quantum computing.

Definition 20.12 (Quantum Parallelism)**.** Consider a circuit that takes x as input and returns $f(x)$ as output. Normally, passing in an input, sees the function applied once, and one output produced. Using quantum gates, such as a Fredkin gate, if x is a quantum register with a superposition of states, it is passed as input and the function is applied once. But the function operates on all of the states of the quantum register, returning output that contains information about the function applied to all states.

The parallelism that can be achieved is the promising feature of quantum computing. The following example aims to illustrate the idea.

Example 20.3 (Classical Function)**.** Consider the function $f(x) = 3x \bmod 8$. Assume we want to calculate all possible answers for $x = 0, 1, 2, \ldots, 7$. With a classical computer we would have a 3-bit input to a circuit that calculates $f(x)$, i.e. performs the modular multiplication. To find all possible answers we would calculate $f(0) = 0$, $f(1) = 3$, $f(2) = 6$, $f(3) = 1$, $f(4) = 4$, $f(5) = 7$, $f(6) = 2$, and $f(7) = 5$. The function/circuit is applied 8 times.

The above example used decimal values, but also consider their binary values, i.e. the function is applied to 8 values: 000, 001, 010, 011, 100, 101, 110 and 111.

Example 20.4 (Quantum Function)**.** Now consider the same function, $f(x) = 3x \bmod 8$, but implemented with a quantum circuit. We initialise a quantum register with 3 qubits. This register is in a superposition of 8 states at once: 000, 001, 010, 011, 100, 101, 110 and 111. The quantum register is input to the circuit. The output register will have 3 qubits in a superposition that contains *all 8 answers*. By applying the function/circuit only once, we obtain an output that has information about all 8 answers. This represents a speedup of a factor of 8 compared to the classical example!

While this a contrived example with many real flaws, it aims to demonstrate that quantum parallelism is achieved by the fact that the quantum calculation is one all states of the quantum register, rather than just a single value as in classical computing.

You should already recognise a problem with the above example. While the output quantum register contains qubits in a superposition that contains information about all 8 answers, when we measure the output register we get just one of those answers with some probability, i.e. the measurement problem. If the probabilities were all equal, i.e. 12.5%, then when we measure the output we would get a value of 000 with probability 12.5%. If we did it again, we may get 011 with probability 12.5%. So the answer is essentialy useless to us; we'd need to calculate 8 times, resulting in the same effort as a classical computer. Quantum algorithms are designed so that the weights/probabilities of the output do give the "correct" answer with high probability.

Definition 20.13 (Quantum Algorithm)**.** A quantum algorithms are usually a combination of classical algorithms/computations and quantum computations. First preprocessing is performed using classical techniques. Then the input quantum register is prepared, a quantum calculation performed, and output quantum register is measured. There may be some post-processing of the result with classical techniques. If the result is as desired, then exit, otherwise repeat the process. Repetition is usually needed due to both errors in quantum calculations and the probabilistic nature of the result.

The main point to note is that "quantum" algorithms actually are a hybrid of classical algorithms and quantum calculations.

The following are two examples of quantum algorithms which are relevant to cryptography.

Definition 20.14 (Grover's Search Algorithm)**.** Consider a database of N unstructured data items (e.g. not sortable). Search is performed by applying a boolean function on input that returns true if correct answer. Classical search takes $\mathcal{O}(N)$ applications of function. Grover's quantum search algorithm takes $\mathcal{O}(\sqrt{N})$ applications of function.

Grover's search algorithm can be used for a brute-force attack. For example with a symmetric key cipher, assume we have a function that decrypts the ciphertext and returns true of the obtained plaintext is correct.

Key length [bits]	Classical	Quantum
64	2^{64}	$\sqrt{2^{64}} = 2^{32}$
128	2^{128}	$\sqrt{2^{128}} = 2^{64}$
256	2^{256}	$\sqrt{2^{256}} = 2^{128}$
512	2^{512}	$\sqrt{2^{512}} = 2^{256}$

Table 20.1: Worst Case Brute Force Attempts with Classical and Quantum Algorithms

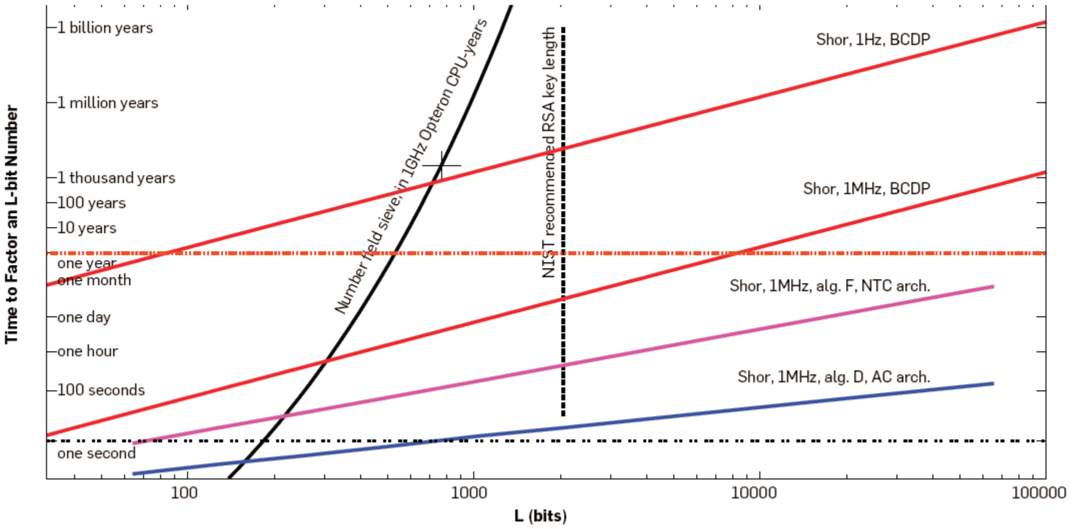
Table 20.1 shows worst case number of attempts a brute-force attack on a key , using either a classical algorithm or Grover's quantum search algorithm. Note that $\sqrt{2^N} = 2^{N/2}$. While the quantum algorithm produces a significant speedup, with regards to protecting symmetric key ciphers against brute force attacks using quantum computers, an easy solution is to double the key length. That is, if a 128-bit key was recommended as secure against brute force attacks using today's classical computers, then to be secure against brute force attacks with future quantum computers, use a 256-bit key. While using a double length key incurs a performance drop for AES, it is not so substantial that makes AES too slow to use, and does not require a new algorithm design.

Now let's look at the promising benefits of quantum computing regarding breaking ciphers, factoring numbers. Recall that integer factorisation is a problem that public key algorithms, such as RSA, are built around. That is, the security of RSA depends on the difficulty of integer factorisation. Let's look at how the best known algorithms on classical and quantum computers perform (we will not look at how those algorithms actually work).

Definition 20.15 (Integer Factorisation with General Number Field Sieve)**.** Given an integer N, find its prime factors. A general number field sieve on classical computer takes subexponential time, about $2^{\mathcal{O}(N^{1/3})}$.

Definition 20.16 (Integer Factorisation with Schor's Algorithm)**.** Given an integer N, find its prime factors. Shor's algorithm on a quantum computer takes polynominal time, about $\log N$.

The paper A Blueprint For Building a Quantum Computer by Rodney Van Meter and Clare Horsman, published in Communications of the ACM, October 2013, has compared the speeds for specific implementations of algorithms on classical and quantum computers. Note that the following results are mainly theoretical, estimating the performance based on several actual measurements with smaller numbers.



Credit: Figure 1 from A Blueprint For Building a Quantum Computer by Van Meter and Horsman, Communications of the ACM, Oct 2013.

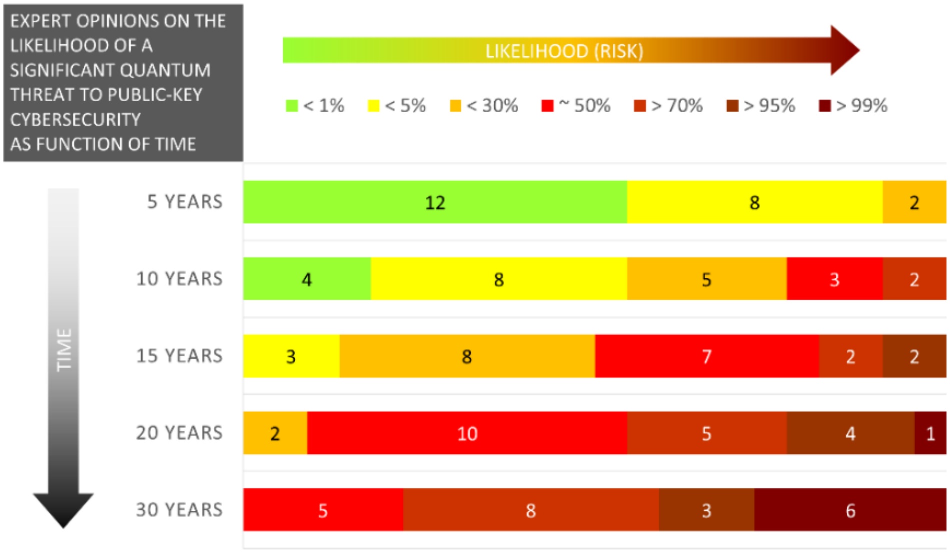
Copyright by Van Meter and Horsman and ACM.

Figure 20.1: Scaling the classical number field sieve (NFS) vs. Shor's quantum algorithm for factoring

Figure 20.1 shows estimated time to factor a L-bit number. The number field sieve on the solid black line is using a classical computer. The cross on that line is for the point of L=768 bits and 3300 CPU years. The NIST recommended key length is L=2048 bits. The lines labelled with Shor are using a quantum computer. The four lines for Shor are different algorithms and architectures, as well as different quantum clock speeds (1Hz vz 1MHz).

One way to read the figure is to look at the number of bits that can be factored in 1 year. A 1GHz classical computer using number field sieve could factor a 500 bit number. A quantum computer using Shor's algorithm and with a 1 Hz clock could factor a 80 bit number. But with a 1 MHz clock it could factor a 8000 bit number.

Is it likely that quantum computers will break RSA in the near future? Michele Mosca and Marco Piani, from evolutionQ and the Global Risk Institute, interviewed 22 experts in quantum computing, and one question was about the likelihood that quantum computers being a significant threat to public-key cryptosystems in the future.



Numbers reflect how many experts (out of 22) assigned a certain probability range.

Credit: Quantum Threat Timeline Report, Michele Mosca and Marco Piani, from evolutionQ and the Global Risk Institute, 2019.

Figure 20.2: Likelihood quantum computers significant threat to public-key cryptosystems

Figure 20.2, from the Quantum Threat Timeline Report, shows the opinions of 22 quantum computing experts. Most think quantum computing will not be a threat to public-key cryptosystems in the next 5 years, and more than half, also in the next 10 years. Almost all think there is a 50% or greater chance that quantum computing will threaten RSA in the next 20 years.

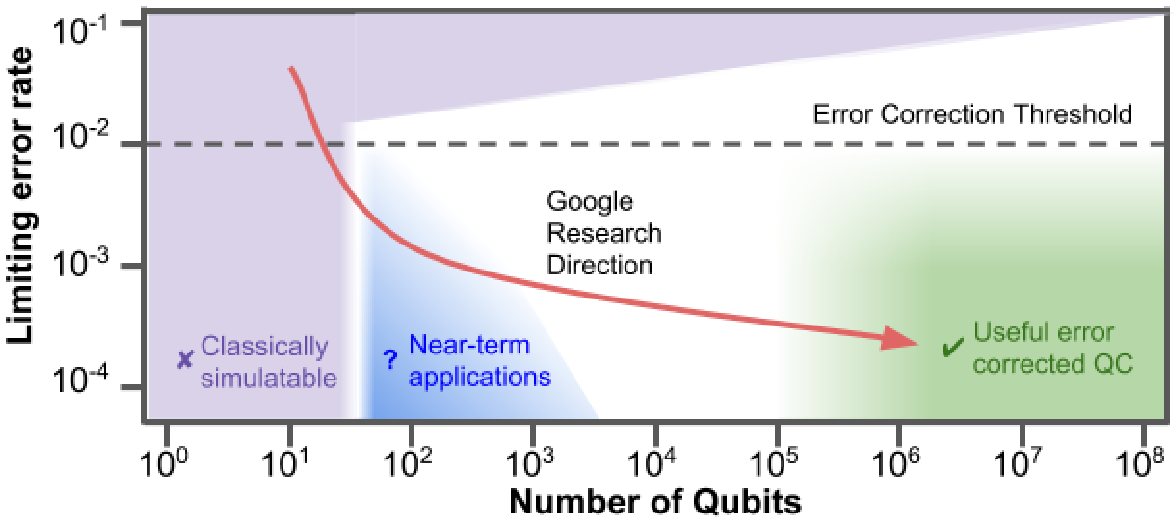
20.3 Issues in Quantum Computing

Definition 20.17 (Decoherence in Quantum Computing)**.** In their coherent state, qubits are described as a superposition of states. The loss of coherence (i.e. decoherence) means the qubits revert to their "classical" basis states. They no longer exhibit the unique quantum properties. Decoherence times vary for different system; for example IBM quantum computers about 100 μs.

Increasing the time that qubits can hold their coherent state is one practical aim of quantum computing. See the T2 column in the Quantum Computing Report for example values.

Definition 20.18 (Errors in Quantum Computing)**.** Errors frequently occur due to various reasons including: decay of individual qubits; environmental defects that impact multiple qubits; interference between qubits and other systems; accidental measurement of qubits; and even loss of qubits. Significant research effort is on designing error correcting schemes.

Error correcting schemes introduce an overhead, and one concern is that the overhead needed to deal with errors may mean quantum computing does not produce significant advantages over classical computing.



Credit: Google Research, A Preview of Bristlecone, Google's New Quantum Processor

Figure 20.3: Quantum error rates vs qubits and intended direction of Google Quantum Research

Figure 20.3, taken from A Preview of Bristlecone, Google's New Quantum Processor by Google Quantum AI Lab, illustrates the conceptual relationship between error rates and qubits. The error correction threshold indicates error rates below this are needed for error correction to work.

Definition 20.19 (Cooling)**.** For qubits to maintain coherence, quantum circuits need to be very cold, approaching 0 Kelvin or -273 C.

Finally, what are the quantum computers available today?

- For more detailed comparison see the Quantum Computing Report
- Google: Sycamore 53-qubit (2019)
- IBM: 5- and 16-qubit machines available for free; 20-qubit machine available via cloud; 53-qubit machine (2019)
- Rigetti: Aspen-7 28-qubits (2019)
- D-Wave systems: 2000Q has 2048-qubits, however using different technology (quantum annealing) that cannot be used to solve Shor's algorithm

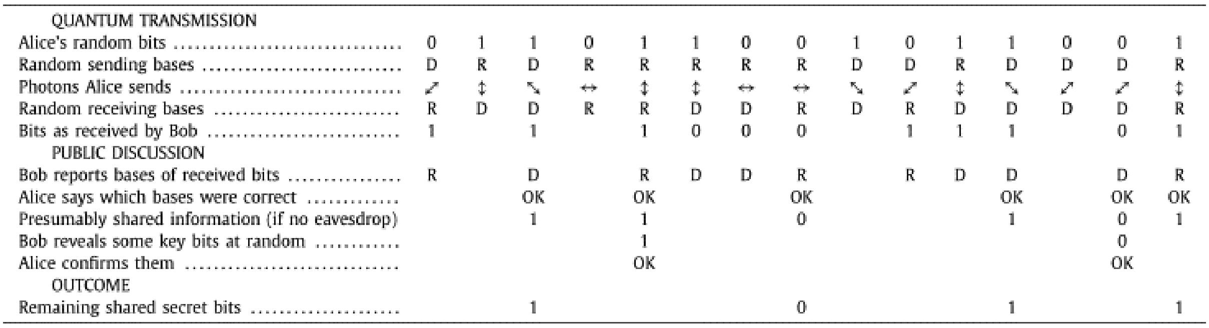
20.4 Quantum Cryptography

Definition 20.20 (Quantum Cryptography)**.** Quantum cryptography refers to techniques that apply principles of quantum systems to build cryptographic mechanisms. The most widely technique is quantum key distribution. Others approaches often involve agreements between parties that do not trust each other.

Note that while quantum computers can be used to break cryptographic mechanisms (e.g. using Schor's algorothm), quantum cryptography is separate topic of quantum systems that is about creating cryptographic mechanisms. Quantum cryptographic mechanisms will use quantum computers.

Definition 20.21 (Quantum Key Distribution (informal))**.** The aim of Quantum Key Distribution (QKD) is for two parties to exchange a secret key (similar to DHKE). A chooses random bits, as well as corresponding random modification of states (called *sending basis*). Applied together using a fixed scheme, A generates and sends photons in quantum states. B chooses own random *measuring basis* and measures the photons. A then informs B their sending basis, and allowing B to recognise which of the measured photons to consider (i.e. those where the measuring basis and sending basis match). B uses the resulting bits as a secret key, however only after confirming with A that there are no errors in the key (e.g. sending a challenge encrypted with the key).

For a formal explanation of QKD, with an example see: https://www.cse.wustl.edu/~jain/cse571-07/ftp/quantum/ or the original paper on one scheme BB84 at https://doi.org/10.1016/j.tcs.2014.05.025.



QUANTUM TRANSMISSION															
Alice's random bits	0	1	1	0	1	1	0	0	1	0	1	1	0	0	1
Random sending bases	D	R	D	R	R	R	R	R	D	D	R	D	D	D	R
Photons Alice sends	⤢	↕	⤡	↔	↕	↕	↔	↔	⤡	⤢	↕	⤡	⤢	⤢	↕
Random receiving bases	R	D	D	R	R	D	D	R	D	R	D	D	D	D	R
Bits as received by Bob	1		1		1	0	0	0		1	1	1		0	1
PUBLIC DISCUSSION															
Bob reports bases of received bits	R		D		R	D	D	R		R	D	D		D	R
Alice says which bases were correct			OK		OK			OK				OK		OK	OK
Presumably shared information (if no eavesdrop)			1		1			0				1		0	1
Bob reveals some key bits at random					1									0	
Alice confirms them					OK									OK	
OUTCOME															
Remaining shared secret bits			1					0				1			1

Credit: Bennett and Brassard, Quantum cryptography: Public key distribution and coin tossing, Theoretical Computer Science, Dec 2014,

Copyright Elsevier.

Figure 20.4: Example of BB84 Quantum Key Distribution

Figure 20.4 is taken from the original 1984 article by Bennet and Brassard, which was re-published by Elsevier in the journal Theoretical Computer Science in 2014. BB84 is a scheme still used for quantum key distribution. The paper, in section III, has a nice explanation of the protocol.

Definition 20.22 (QKD security (informal))**.** An attacker C tries to learn the secret key between A and B, without A or B knowing. Therefore the attacker has to measure the photons sent by A. However, as the photons are a superposition of states, when C measures them, they are changed. As a result, B will receive changed photons, and when they check the secret key with A, the check will fail.

The security of quantum key distribution depends on that measurement problem, i.e. that measuring a quantum superposition state, changes the state. The attacker cannot measure the communications between A and B without changing the communications. It is easy for A and B to recognise if the communications have been changed.

20.5 Cryptography in the Quantum Era

Given that attacks on existing, widely-used ciphers such as RSA may be possible in the (long-term) future using quantum computers, researchers and standardisation organisations are working on ciphers that are resistant to attacks with quantum technologies. This is referred to as *post-quantum cryptography*.

- NIST Post-Quantum Cryptography project called for proposals on quantum-resistant public key cryptography algorithms
  - Digital signatures, public-key encryption, key exchange
  - 69 submissions in round 1 (2017)
  - 26 algorithms in round 2 (2019)
  - 7 finalists in round 3 (2020)
  - Plan to standardise in 2022/2023
- Open Quantum Safe has open-source software for prototyping quantum-resistant cryptography, including forks of OpenSSL, OpenSSH and OpenVPN

Part VIII

Additional Resources

Appendix A

Cryptography Assumptions and Principles

Cryptography is a large, complex topic. However even if the details are not understood, we can still apply concepts from cryptography to design secure systems. This chapter lists some common assumptions that are made about cryptographic techniques as well as some principles that are used in designing secure systems. Although in theory the assumptions do not always hold, they are true in many practical situations (and when they are not true, it will be made clear).

A.1 Assumptions

A.1.1 Encryption

A1. Symmetric key cryptography is also called conventional or secret-key cryptography.

A2. Public key cryptography is also called asymmetric key cryptography.

A3. In symmetric key crypto, the same secret key, K, is used for encryption, E(), and decryption, D(). The secret is shared between two entities, i.e. K_{AB}.

A4. In public key crypto, there is a pair of keys, public (PU) and private (PR). One key from the pair is used for encryption, the other is used for decryption. Each entity has their own pair, e.g. (PU_A, PR_A).

A5. Encrypting plaintext (or a message), P or M, with a key, produces ciphertext C, e.g. $C = \mathrm{E}(K_{AB}, P)$ or $C = \mathrm{E}(PU_A, M)$.

A6. Decrypting ciphertext with the correct key will produce the original plaintext. The decrypter will be able to recognise that the plaintext is correct (and therefore the key is correct). E.g. $P = \mathrm{D}(K_{AB}, C)$ or $M = \mathrm{D}(PR_A, C)$.

A7. Decrypting ciphertext using the incorrect key will *not* produce the original plaintext. The decrypter will be able to recognise that the key is wrong, i.e. the decryption will produce unrecognisable output.

File: crypto/secassume.tex, r1697

A.1.2 Knowledge of Attacker

A8. All algorithms used in cryptography, e.g. encryption/decryption algorithms, hash functions, are public.

A9. An attacker knows which algorithm is being used, and any public parameters of the algorithm.

A10. An attacker can intercept any message sent across a network.

A11. An attacker does not know secret values (e.g. symmetric secret key K_{AB} or private key PR_A).

A12. Brute force attacks requiring greater than 2^{80} operations are impossible.

A.1.3 Authentication with Symmetric Key and MACs

A13. An entity receiving ciphertext that successfully decrypts with symmetric secret key K_{AB} knows that the original message has not been modified and that it originated at one of the owners of the secret key (i.e. A or B).

A14. An entity receiving a message with attached MAC that successfully verifies, knows that the message has not been modified and originated at one of the owners of the MAC secret key.

A.1.4 Hash Functions

A15. A cryptographic hash function, H(), takes a variable sized input message, M, and produces a fixed size, small output hash, h, i.e. $h = \mathrm{H}(M)$.

A16. Given a hash value, h, it is impossible to find the original message M.

A17. Given a hash value, h, it is impossible to find another message M' that also has a hash value of h.

A18. It is impossible to find two messages, M and M', that have the same hash value.

A.1.5 Digital Signatures

A19. A digital signature of a message M is the hash of that message encrypted with the signers private key, i.e. $S = \mathrm{E}(PR, \mathrm{H}(M))$

A20. An entity receiving a message with an attached digital signature knows that that message originated by the signer of the message.

A.1.6 Key Management and Random Numbers

A21. A secret key can be exchanged between two entities without other entities learning its value.

A22. Any entity can obtain the correct public key of any other entity.

A23. Pseudo-random number generators (PRNG) can generate effectively true random numbers.

A.2 Principles

P1. *Experience*: Algorithms that have been used over a long period are less likely to have security flaws than newer algorithms.

P2. *Performance*: Symmetric key algorithms are significantly faster than public key algorithms.

P3. *Performance*: The time to complete a cryptographic operation is linearly proportional with the input data size.

P4. *Key Distribution*: Keys should be distributed using automatic means.

P5. *Key Re-use*: The more times a key is used, the greater the chance of an attacker discovering that key.

P6. *Multi-layer Security*: Using multiple overlapping security mechanisms can increase the security of a system.

Appendix B

Data Formats

Information that is to be secured can be represented in a variety of data formats. In this chapter we list key data formats used throughout the book, and more generally in cryptography. We demonstrate tools for manipulating the data.

B.1 Common Data Formats

B.1.1 English Alphabet

A character set of the 26 letters in the English alphabet:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

Unless otherwise stated, case insensitive. A case sensitive variation would have 52 characters in the set. Other variations are possible, where additional characters are included (e.g. digits, punctuation) or different languages are used.

Alphabetical ordering is used, and often the letters are mapped to integers, starting at a = 0.

Primarily seen in classical ciphers.

B.1.2 Printable Keyboard Characters

A character set consisting of the characters printable from the keys on a typical keyboard. On US/English keyboards, usually 94 characters:

- 26 uppercase English letters
- 26 lowercase English letters
- 10 digits
- 32 punctuation characters (see your keyboard)

Keys such as SPACE, TAB and ENTER are usually not considered printable.

Primarily seen in applications dealing with user input, e.g. passwords.

File: crypto/formats.tex, r1766

B.1.3 Binary Data

In modern systems, all data is represented as binary values. This includes text, documents, images, applications, audio and video. There are different encodings to map these data into binary (some of which are described in this chapter).

In this book, when referring to sequence of bits, the 1st bit refers to the left most bit in the sequence. (In some cases, bits are indexed starting at 0, e.g. the 0th bit, the 1st bit; it is made clear when this is the case). For example, for the sequence `01001111`, the 1st bit is 0, the 2nd bit is 1, the 3rd bit is 0 and the last (8th) bit is 1. Also, the most significant bit is the left most bit. In the previous example, the 2nd bit has the decimal value of 64, and the last (8th) has the decimal value of 1.

Note that encoding and decoding is not equivalent to encryption and decryption. That is, encoding (from say ASCII to Base64) does not provide any significant security value as there is not key involved. Even if an attacker did not know the encoding used, they could easily try all possible encodings.

B.1.4 ASCII

American Standard Code for Information Interchange (ASCII) is a common standard for representing keyboard/computer characters in a digital format. Also referred to as the International Reference Alphabet and a subset of Unicode, there are 128 characters in the ASCII character set. Section B.1 shows the mappings to decimal values, while Section B.2 shows the mapping to 7-bit binary values (take the 3 bits from the column and then the 4 bits from the row).

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	–	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Figure B.1: International Reference Alphabet, or ASCII, Table in Decimal

While ASCII can be represented in 7-bits, it is commonly used in computer files as 8-bit values, where the 1st bit is always a binary 0. For example, uppercase A is binary `01000001`.

Ordering is by the numerical value, e.g. `!` comes before `A`, which comes before `a`.

You can see the standard 94 printable keyboard characters from `!` through to `~`.

	First 3 bits							
Last 4 bits	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	–	o	DEL

Figure B.2: International Reference Alphabet, or ASCII, Table in Binary

B.1.5 Hexadecimal

A character set with 16 characters:
0 1 2 3 4 5 6 7 8 9 A B C D E F

When communicating binary data (to humans), it is sometimes represented in hexadecimal as it uses four times less characters (4 bits per character), and has less chance of reading/writing errors.

Examples of using hexadecimal to illustrate binary data includes: secret keys, public key pair values, very large numbers (e.g. large primes), ciphertext, and addresses.

B.1.6 Base64

An alternative to hexadecimal representation of binary data is using Base64 encoding. Base64 is a character set of 64 characters:

- 26 uppercase English letters
- 26 lowercase English letters
- 10 digits
- 2 punctuation characters which are + and /

The = character is used to indicate padding (and is not part of the 64 characters). See online resources for an explanation of padding.

Base64 maps 6 bits to a character and therefore is more concise than hexadecimal. It is often used when communicating binary data in text-based protocols in networks (e.g. including binary data in a HTML page or email).

B.2 Conversions using Linux

In Linux, `xxd` is useful for viewing text files (containing ASCII) in binary and hexadecimal. See Section 3.1.1.

For Base64, the command `base64` can be used:

```
$ echo -n "This is a message." > data.txt
$ xxd data.txt
00000000: 5468 6973 2069 7320 6120 6d65 7373 6167 This is a messag
00000010: 652e                                e.
$ base64 data.txt
VGhpcyBpcyBhIG1lc3NhZ2Uu
$ base64 data.txt > data.b64
$ base64 -d data.b64
This is a message.$
```

To convert ASCII characters to their decimal value, in a Linux Bash terminal you can use `printf` (newlines have been added below to make the output clearer):

```
$ printf '%d' "'A"
65
$ printf '%d' "'a"
97
$ printf '%d' "'!"
33
$ printf '%d' "'~"
126
```

It is a little more cumbersome in the opposite direction:

```
$ printf "\\$(printf '%03o' "65")"
A
$ printf "\\$(printf '%03o' "97")"
a
$ printf "\\$(printf '%03o' "33")"
!
$ printf "\\$(printf '%03o' "126")"
~
```

You are advised to simply lookup the table or find another tool, rather than use the Bash commands as above.

B.3 Conversions using Python

There are different ways to convert between varying formats in Python. The following code shows some examples. The code is also available in the Steve's Workshops GitHub repository. The code below is version 4c0faec. An example of the output from running the conversion functions follows the code.

Listing B.1: Demonstration of converting between different formats in Python

```
'''
Convert data between different formats. No (or very little) error checking
is performed. You need to make sure the input data for the conversion is
in the format specified.
'''

import base64
import logging
logger = logging.getLogger("Conversions")

def bytes_to_text(b):
    return b.decode('utf-8')

def text_to_bytes(s):
    return s.encode('utf-8')

def bytes_to_base64(b):
    return bytes_to_text(base64.b64encode(b))

def base64_to_bytes(b64):
    return base64.b64decode(b64)

def bytes_to_hex(b):
    return b.hex()

def hex_to_bytes(h):
    return bytes.fromhex(h)

def base64_to_text(b64):
    return bytes_to_text(base64_to_bytes(b64))

def base64_to_hex(b64):
    return bytes_to_hex(base64_to_bytes(b64))

def text_to_base64(s):
    return bytes_to_base64(text_to_bytes(s))

def hex_to_base64(h):
    return bytes_to_base64(hex_to_bytes(h))

def text_to_hex(s):
    return bytes_to_hex(text_to_bytes(s))

def hex_to_text(h):
    return bytes_to_text(hex_to_bytes(h))

def text_to_list(s):
    return list(s)

def list_to_text(l):
    return "".join(l)

```

```
def hex_to_binary(h):
    return bin(int(h,16))[2:]

def binary_to_hex(bi):
    return hex(int(bi,2))[2:]

def binary_to_bytes(bi):
    return hex_to_bytes(binary_to_hex(bi))

def bytes_to_binary(b):
    return hex_to_binary(bytes_to_hex(b))

def text_to_binary(s):
    return bytes_to_binary(text_to_bytes(s))

def binary_to_text(bi):
    return bytes_to_text(binary_to_bytes(bi))

def base64_to_binary(b64):
    return bytes_to_binary(base64_to_bytes(b64))

def binary_to_base64(bi):
    return bytes_to_base64(binary_to_bytes(bi))


def letter_to_number(c, charset="lowercase"):
    '''
    Convert a single character into a number
    Converts a -> 0, b -> 1, c -> 2, ... or
    if uppercase A -> 0, B -> 1, C -> 2, ...
    '''

    if charset == "uppercase":
        return ord(c) - 65
    else:
        return ord(c) - 97

def number_to_letter(n, charset="lowercase"):
    '''
    Convert a number into a single character
    See char_to_num(c) - this is the opposite
    '''

    if charset == "uppercase":
        return chr(n + 65)
    else:
        return chr(n + 97)

def text_to_numbers(text, charset="lowecase"):
    '''
    Convert a string into a list of numbers
    :Example:
     - input: str = "abc"
     - output: list = [0, 1, 2]
    '''

    return [letter_to_number(c, charset) for c in text]
```

```

def numbers_to_text(nums, charset="lowercase"):
    '''
    Convert a list of numbers into a string
    See text_to_nums(text) - this is the opposite
    '''

    return ''.join([num_to_char(n, charset) for n in nums])


if __name__=='__main__':
    import sys
    import argparse

    # Process command line arguments
    parser = argparse.ArgumentParser(
        description="Convert␣between␣different␣formats␣for␣cryptography",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog='''
example (command-line):
$ python conversions.py
''')
    parser.add_argument("-l", "--log",
        choices=["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"])
    args = parser.parse_args()

    # Enable logging based on command line input
    if args.log is None:
        numeric_log_level = logging.ERROR
    else:
        numeric_log_level = getattr(logging, args.log.upper(), None)
        if not isinstance(numeric_log_level, int):
            raise ValueError('Invalid␣log␣level:␣%s' % args.log)
    logging.basicConfig(level=numeric_log_level)

    data1_str = "Hello"
    data1_bytes = text_to_bytes(data1_str)
    data1_b64 = text_to_base64(data1_str)
    data1_hex = text_to_hex(data1_str)
    data1_bin = text_to_binary(data1_str)
    data1_list = text_to_list(data1_str)

    print("Converting␣Text␣to␣...")
    print("␣␣␣Text:" + str(data1_str))
    print("␣␣␣Bytes␣:" + str(data1_bytes))
    print("␣␣␣Base64:" + str(data1_b64))
    print("␣␣␣Hex␣␣␣:" + str(data1_hex))
    print("␣␣␣Binary:" + str(data1_bin))
    print("␣␣␣List␣␣:" + str(data1_list))

    data2_b64 = "SGVsbG8="
    data2_bytes = base64_to_bytes(data2_b64)
    data2_str = base64_to_text(data2_b64)
    data2_hex = base64_to_hex(data2_b64)
    data2_bin = base64_to_binary(data2_b64)

    print("Converting␣Base64␣to␣...")
    print("␣␣␣Text:" + str(data2_str))
    print("␣␣␣Bytes␣:" + str(data2_bytes))
    print("␣␣␣Base64:" + str(data2_b64))
    print("␣␣␣Hex␣␣␣:" + str(data2_hex))
    print("␣␣␣Binary:" + str(data2_bin))

    data3_hex = "48656c6c6f"
    data3_bytes = hex_to_bytes(data3_hex)
    data3_str = hex_to_text(data3_hex)
    data3_b64= hex_to_base64(data3_hex)
    data3_bin = hex_to_binary(data3_hex)

    print("Converting␣Hex␣to␣...")
    print("␣␣␣Text:" + str(data3_str))
    print("␣␣␣Bytes␣:" + str(data3_bytes))
    print("␣␣␣Base64:" + str(data3_b64))
    print("␣␣␣Hex␣␣␣:" + str(data3_hex))
    print("␣␣␣Binary:" + str(data3_bin))

    data4_chr = 'c'
    data4_num = letter_to_number(data4_chr)
    data5_str = "hello"
    data5_nums = text_to_numbers(data5_str)

    print("Letter␣" + data4_chr + "␣is␣" + str(data4_num))
    print("Text␣" + data5_str + "␣is␣" + str(data5_nums))
```

```
sgordon@chilli:~/git/workshops/python/demos$ python3 conversions.py
Converting Text to ...
   Text:Hello
   Bytes :b'Hello'
   Base64:SGVsbG8=
   Hex   :48656c6c6f
   Binary:1001000011001010110110001101100011011111
   List  :['H', 'e', 'l', 'l', 'o']
Converting Base64 to ...
   Text:Hello
   Bytes :b'Hello'
   Base64:SGVsbG8=
   Hex   :48656c6c6f
   Binary:1001000011001010110110001101100011011111
Converting Hex to ...
   Text:Hello
   Bytes :b'Hello'
   Base64:SGVsbG8=
   Hex   :48656c6c6f
   Binary:1001000011001010110110001101100011011111
Letter c is 2
Text hello is [7, 4, 11, 11, 14]
```

Appendix C

Organisations and People in Cryptography

C.1 Organisations in Cryptography and Security

C.1.1 National Institute of Standards and Technology

National Institute of Standards and Technology (NIST) https://www.nist.gov/

C.1.2 International Association for Cryptologic Research

International Association for Cryptologic Research (IACR) https://www.iacr.org/

C.1.3 Australian Signals Directorate

Australian Signals Directorate (ASD) https://www.asd.gov.au/

C.1.4 National Security Agency

National Security Agency (NSA) https://www.nsa.gov/

C.1.5 Government Communications Headquarters

Government Communications Headquarters (GCHQ) https://www.gchq.gov.uk/

C.1.6 Institute of Electrical and Electronics Engineers

Institute of Electrical and Electronic Engineers (IEEE) https://www.ieee.org/

C.1.7 Internet Engineering Task Force

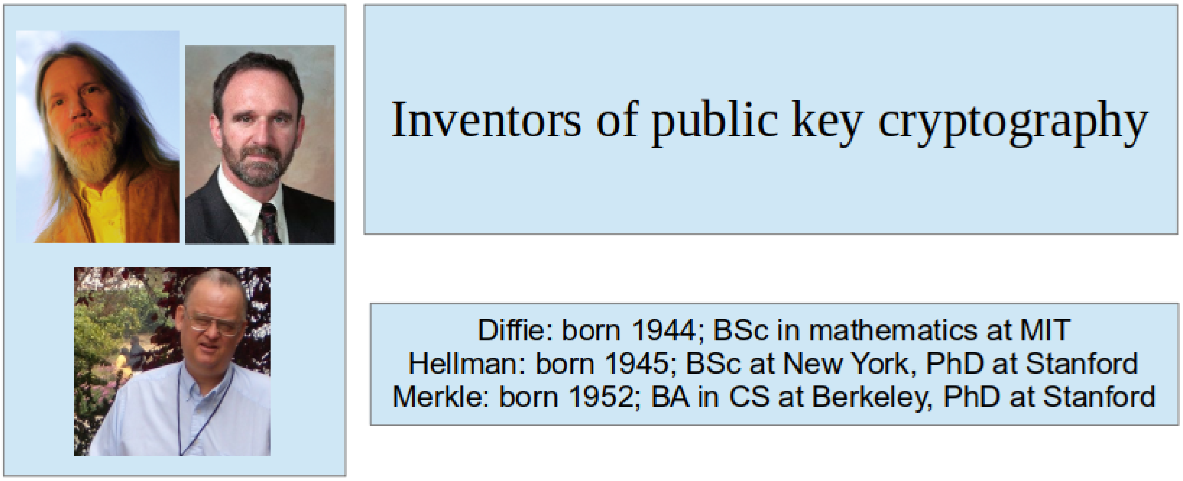
Internet Engineering Task Force (IETF) https://www.ietf.org/

File: crypto/orgs.tex, r1805

C.2 People in Cryptography and Security

This section lists a selection of people that have made important and/or interesting contributions to security and cryptography. The list does not attempt to be definitive, and some key people are omitted. The biographies are brief, with information mainly taken from Wikipedia. Again, the biographies doe not attempt to cover all aspects of the person's life, but rather to trigger your interest to explore the backgrounds of these and other people further.

C.2.1 Diffie, Hellman and Merkle



Inventors of public key cryptography

Diffie: born 1944; BSc in mathematics at MIT
Hellman: born 1945; BSc at New York, PhD at Stanford
Merkle: born 1952; BA in CS at Berkeley, PhD at Stanford

Figure C.1: Diffie, Hellman and Merkle

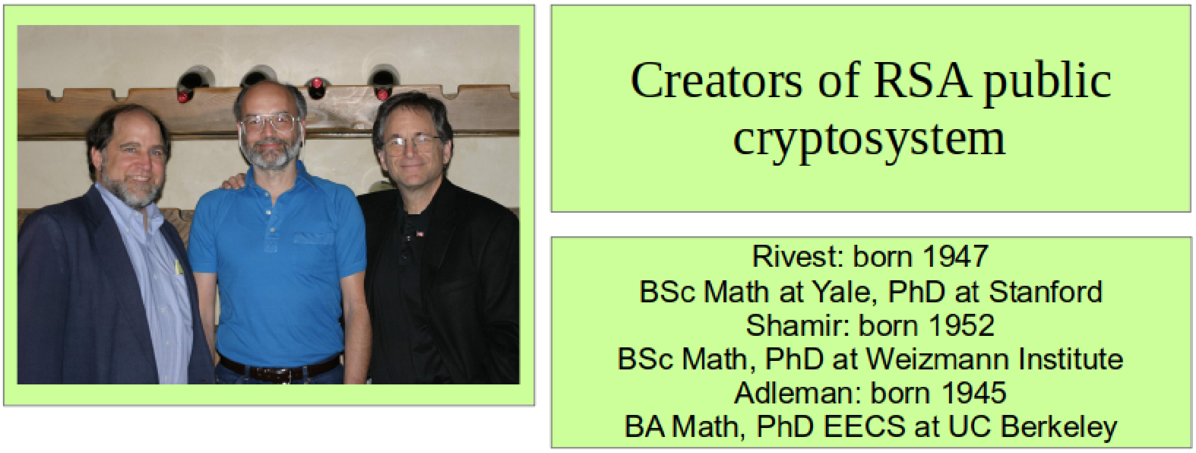
While studying his Bachelor degree in computer science in 1974, Ralph Merkle developed a set of puzzles that allowed two users to agree upon a shared secret key by exchanging messages over an unsecure channel, even if they have no common secrets known beforehand. This was unique as up until then, as it was normally assumed users must manually exchange a secret before than can send messages. Ralph continued his studies in a PhD with Martin Hellman as his adviser.

In 1976 Whitfield Diffie and Martin Hellman used Merkle's scheme as motivation for their own, improving the security by basing the problem of the attacker on solving discrete logarithms (Merkle's puzzles only involved quadratic complexity problems, much easier than discrete logarithms). Their scheme, called Diffie-Hellman key exchange, was the first secure example of public key cryptography. It is still in use today, in particular in TLS (e.g. when you SSH into another computer).

In the 1990's it was announced that Clifford Cocks and others at GCHQ had designed similar public key cryptography concepts earlier than Merkle, Diffie and Hellman.

C.2.2 Rivest, Shamir and Adleman

At MIT in the 1970's, Rivest, Shamir and Adleman created the RSA algorithm for public key cryptography. The algorithm defines how a user creates a public-private key pair, and can then encrypt a message using one of the keys such that it can only be successfully



Creators of RSA public cryptosystem

Rivest: born 1947
BSc Math at Yale, PhD at Stanford
Shamir: born 1952
BSc Math, PhD at Weizmann Institute
Adleman: born 1945
BA Math, PhD EECS at UC Berkeley

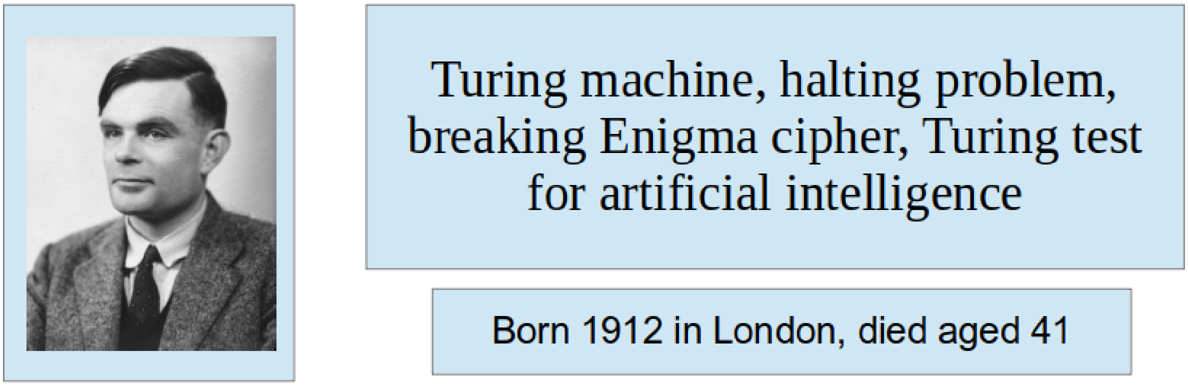
Figure C.2: Rivest, Shamir and Adleman

be decrypted by the other key of the pair. The strength of RSA is based on the difficult to factor large numbers into their prime factors.

Although their were other public key algorithms developed, before RSA symmetric key encryption was primarily used in practice. With RSA patented, Rivest, Shamir and Adleman co-founded RSA Security to commercialise the use of the algorithm. In 2006 it was acquired by EMC for $US2 billion. RSA is mainly used for digital signatures and authentication tokens. Verisign was formed as a spin-off company from RSA Security that used the algorithm to sign digital certificates.

Rivest, Shamir and Adleman continue their cryptography research. Rivest developed ciphers RC2, RC4, RC5 and RC6 and hash functions MD2, MD4, MD5 and MD6; Shamir discovered differential crytpanalysis; Adleman is a leader of DNA computing and coined the term ’computer virus’.

C.2.3 Alan Turing



Turing machine, halting problem, breaking Enigma cipher, Turing test for artificial intelligence

Born 1912 in London, died aged 41

Figure C.3: Alan Turing

In 1934 Alan Turing obtained a Bachelor degree in Mathematics at King’s College, Cambridge. He continued there as a researcher and in 1936 published his famous paper that: presented a Turing machine; provided that the halting problem is undecidable;

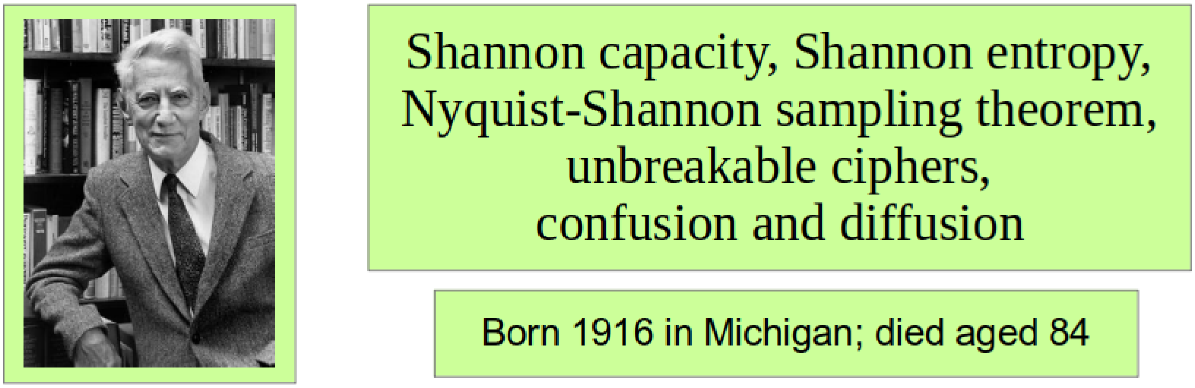
and therefore proved that there is no solution for the Entscheidungsproblem ("decision problem").

Turing then worked at Princeton, obtaining his PhD in 1938, which introduced ordinal logic and the computing oracle, which has been highly influential in computational complexity theory.

In 1938 Turing returned to the England, and during World War II worked for the British code breaking organisation (which is now GCHQ) in Bletchley Park. He made major contributions to breaking the Enigma cryptosystem used by Germans, as well as developing a secure voice scrambler and using statistical techniques to break codes. In 1948 Turing lead the development of one of the first computers. As a contribution to artificial intelligence, he also developed the Turing test, a way to determine if a machine is "intelligent". He also developed LU decomposition, a method used to solve matrix equations.

Turing was convicted and chemically castrated for being homosexual in 1952. He committed suicide in 1954.

C.2.4 Claude Shannon



Shannon capacity, Shannon entropy,
Nyquist-Shannon sampling theorem,
unbreakable ciphers,
confusion and diffusion

Born 1916 in Michigan; died aged 84

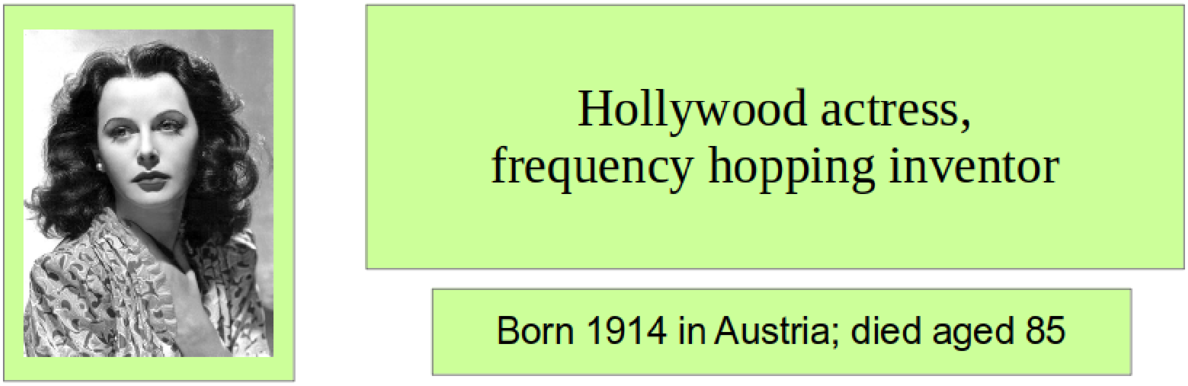
Figure C.4: Claude Shannon

After obtaining bachelor degrees in electrical engineering and mathematics at the University of Michigan, Claude Shannon studied a Masters at MIT where he applied Boolean algebra to design telephone circuit switches. The ideas presented in his thesis had a significant impact on the design of digital circuits used in computers today. In 1940 Shannon completed his PhD at MIT, applying similar techniques to genetics.

During World War II Shannon worked at Bell Labs and started developing ideas which would become key contributions to communications theory and cryptography. In particular, he investigated the theoretical limits of storing and communication data; this is now known as the field of information theory. Shannon and others developed theorems for the maximum amount of data that can be communicated over a bandwidth limited channel in the presence of noise (Shannon capacity), the average amount of information contained in a message (Shannon entropy), and the rate at which analog signals should be sampled to create accurate digital signals (Nyquist-Shannon sampling theorem). He also proved the one-time pad is unbreakable, that other unbreakable ciphers must have the same characteristics as the OTP, and defined diffusion and confusion to be used to secure practical ciphers.

Shannon and others used principles of information theory to make substantial winnings in Las Vegas casinos and on the stock market.

C.2.5 Hedy Lamarr



Hollywood actress, frequency hopping inventor

Born 1914 in Austria; died aged 85

Figure C.5: Hedy Lamarr

In the early 1930's Hedy Lamarr acted in several movies in Europe, before moving the Hollywood in 1938. She had a leading role in multiple top movies in the 1940's, alongside the most popular actors of the time.

While acting during World War II, Lamar was inspired to contribute to the war effort and worked with George Antheil on inventions. They focussed on remote control torpedoes, in particular how to design communications between a ship and torpedo so that the signal could not jammed. They developed a method of rapidly switching or "hopping" between different frequencies (initially 88 frequencies, matching the number of keys on the piano of Antheil). An attacker would need to transmit on all frequencies to jam the signal, which would require too much power, making the attack impractical. Lamarr and Antheil were granted a US patent in 1942.

Although Lamarr did not commercialise the technique, it started to be used by the US military in the 1960's, and more widely in the 1990's. The concept of frequency hopping serves as the basis of spread spectrum communications used today. It is used in Bluetooth, WiFi and CDMA mobile phones.

Lamarr continued acting, gaining a star on the Hollywood Walk of Fame, as well as being inducted into the Inventors Hall of Fame.

C.2.6 Phil Zimmermann

In 1991 Phil Zimmermann wrote Pretty Good Privacy (PGP), which used public key cryptography for email encryption. PGP encrypts the email message with a symmetric key cipher using a random key, and then encrypts that random key with a public key cipher, such as RSA. The sender uses the receives public key to encrypt the random key. For PGP to be useful, people must have potential destinations public keys. Zimmermann used the web of trust to ensure public keys were valid: the more people that you trust that trust a public key, the more you trust that public key.

At the time, exporting cryptography software from the US was illegal, and Zimmermann was investigated for 3 years. He even published the entire source code in a book;

Creator of PGP, the most widely used email encryption software

Born 1954 in New Jersey
BSc in Computer Science at
Florida Atlantic University

Figure C.6: Phil Zimmermann

the US government were unlikely to stop the exportation of a book that could be legally purchased.

Zimmermann continues activities in security and privacy, developing ZRTP for encrypted real-time VOIP calls, and founding Silent Circle which offers secure text, email and phones.

C.2.7 Other People

- Bruce Schneier
- Ross Anderson
- Daniel J. Bernstein
- Dan Boneh
- Joan Daemen

Appendix D

Versions of this Book

This book is work-in-progress. It is expected errors will be fixed, improvements made and new content added on a regular basis. The intention is that:

- A new major version will be released (if necessary) at the start of each teaching term. That is currently March (03), July (07) and November (11). If no significant updates are made between teaching terms, then a new major version may be skipped. The major versions will be named by year and month, e.g. 20.03, 20.07, 20.11, 21.03.
- Minor versions will be released to fix bugs, typos and formatting issues. They may contain new content (e.g. new chapter or new section), so long as the existing chapters and sections are not re-numbered (e.g. new chapters will be added at the end of the book). Apart from this, they will not contain significant changes to the content. The minor versions will be identified by the Subversion (SVN) revision number on the first page of the book.

Summary of changes between versions are listed below.

Crypto 20.03

`r1671`, 1 March 2020: First public release of the book.

Crypto 22.03

`r1972`, 4 January 2022: Replaced many images with own images or Creative Commons images (e.g. DES, AES, Authentication, Classical chapters); changed slides from 4:3 to 16:9 aspect ratio; several additional examples (e.g. Block Cipher Design Principles); new videos for Encryption and Number Theory chapters.

Index

`/dev/random`, 16
`/dev/urandom`, 15

accounting, 6
additive inverse, 35
asymmetric cryptography, *see* public key cryptography
asymmetric key crypto, *see* public-key crypto
attacks
  brute force, *see* brute force attack
  frequency analysis, *see* frequency analysis
authentication, 6
authorisation, 6
availability, 5

`base64`, 218
`bc`, 13
brute force attack, 48
  example, 48
  monoalphabetic cipher, 51
  Python, 48

Caesar cipher, 45
  definition, 45, 47
  example, 45–47
  Python, 48
CIA, 5
cipher, 7
ciphers
  Caesar, *see* Caesar cipher
  monoalphabetic, *see* monoalphabetic cipher
  Playfair, *see* Playfair cipher
  polyalphabetic, *see* polyalphabetic cipher
ciphertext, 7
`cmp`
  example, 163
confidentiality, 5
congruence, 34
congruent modulo, 34
conventional crypto, *see* symmetric key crypto
cryptanalysis, 7
cryptography, 7
cryptology, 7
`cut`, 12
  example, 15

decryption, 7, 102
`diff`
  example, 102, 116
digital signature, 149
discrete logarithm, 39
divides, 31
divisor, 31

`echo`
  example, 103
encryption, 6, 7, 102
Euler's totient, 33

Feistel network, 67
Fermat's theorem, 37
frequency analysis, 52
  monoalphabetic, 52

gcd(), *see* greatest common divisor
`git`
  example, 20
greatest common divisor, 31

integer factorisation, 41
integrity, 5

key, 7

Linux, 11
  Ubuntu, 11
logarithm, 38

`md5sum`, 16
mod, 34
mode of operation, 103
modular addition, 35
modular arithmetic, 34
modular division, 37
modular exponentiation, 38
modular logarithm
    see discrete logarithm 39
modular multiplication, 36
modular subtraction, 35
modulus, 34
monoalphabetic cipher, 50
    definition, 50
    example, 50
multiplicative inverse, 36

One-Time Pad, 60
OpenSSL, 17
    `dgst`, 149
    `enc`, 102
    `genpkey`, 145, 159
    `help`, 17
    `list`, 18
    padding, 103
    `pkey`, 146
    `pkeyparam`, 160
    `pkeyutl`, 149, 162
    `rand`, 19, 104
    `smime`, 150
    `speed`, 116
    `version`, 18

passwords
    salt, 103
permutation, 50, 67
plaintext, 7
Playfair cipher, 54
    definition, 54, 55
    example, 54, 55
polyalphabetic cipher, 56
    definition, 56
prime factorisation, 33
prime factors, 32
prime number, 32
primitive root, 39
public key cryptography, 131, 145
public-key crypto, 65

pycipher, 20
Python, 20

rail fence cipher, 62
random numbers
    `/dev/random`, 16
    `/dev/urandom`, 15
    $RANDOM, 15
relatively prime, 31
rows columns cipher, 62
RSA, 145

S-DES, 91
security protections, 5
`sha1sum`, 16
`sha256sum`, 16
`sha512sum`, 16
shared-key crypto, *see* symmetric key crypto
Simplified-DES, *see* S-DES
single-key crypto, *see* symmetric key crypto
SPN, 67
substitution, 67
symmetric key crypto, 65

transposition, 61

Ubuntu, *see* Linux

Vernam cipher, 59
Vigenère cipher, 57
    example, 57

`wc`
    example, 103

XOR
    Python, 59
`xxd`, 11
    example, 15, 19, 103, 116