Introduction

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
intro.tex, r1960

Slides for this Book

- 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/`

Cryptography Concepts and Terminology

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
concepts.tex, r1961

Contents

Important Security Protections

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.

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.

Other Common 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.

Scope

- 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

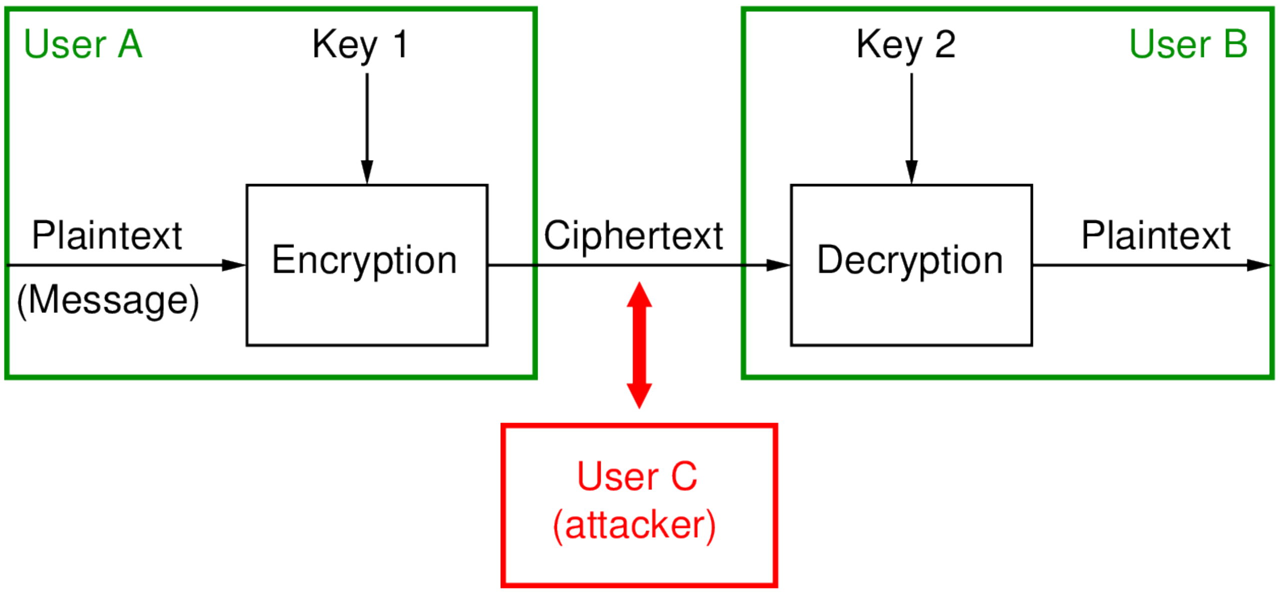
Contents

Encryption for 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.

Model of Encryption for Confidentiality



The figure on slide 8 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 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).

Cryptography Terms

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

Contents

Common Symbols and Notation

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	$A\|\|B$

Software Tools

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
tools.tex, r1962

Statistics for Communications and Security

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 19 Feb 2020,
statistics.tex, r1791

Contents

Properties of Exponentials and Logarithms

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

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

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

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

Properties of Exponentials (example)

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 5 gives the exact or approximate decimal value for b-bit numbers.

Useful Exact and Approximate Values in Binary

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}

Properties of Exponentials with Binary Values (example)

Properties and approximations can be used to perform large calculations:

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

Properties of Logarithms (example)

The number of bits needed to represent a decimal number can be found using logarithms:

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

Contents

Number of Binary Values (definition)

Given an n-bit number, there are 2^n possible values.

Number of Sequence Numbers (example)

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).

Number of IP Addresses (example)

An IP address is a 32-bit value. There are 2^{32} or approximately 4×10^9 possible IP addresses.

Number of Keys (example)

If choosing a 128-bit encryption key randomly, then there are 2^{128} possible values of the key.

Fixed Length Sequences (definition)

Given a set of n items, there are n^k possible k-item sequences, assuming repetition is allowed.

Sequences of PINs (example)

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.

Sequences of Keyboard Characters (example)

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.

Pigeonhole Principle (definition)

If n objects are distributed over m places, and if $n > m$, then some places receive at least two objects.

Pigeonhole Principle on Balls (example)

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.

Pigeonhole Principle on Hash Functions (example)

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.

Contents

Factorial (definition)

There are $n!$ different ways of arranging n distinct objects into a sequence.

Factorial and Balls (example)

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
```

Factorial and English Letters (example)

The English alphabetic has 26 letters, a–z. There are $26! \approx 4 \times 10^{26}$ ways to arrange those 26 letters.

Factorial and Plaintext Messages (example)

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.

Combinations (definition)

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:

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

Number of Pairs (definition)

The number of pairs of items in a set of n items, assuming repetition is not allowed and order doesn't matter, is:

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

Pairs of Coloured Balls (example)

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`.

Pairs of Network Devices (example)

A computer network has 10 devices. The number of links needed to create a full-mesh topology is $10 \times 9/2 = 45$.

Pairs of Key Sharers (example)

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 $50 \times 49/2 = 1,225$.

Contents

Probability of Selecting a Value (definition)

Probability of randomly selecting a specific value from a set of n values is $1/n$.

Probability of Selecting Coloured Ball (example)

There are five coloured balls in a box: red, green, blue, yellow and black. The probability of selecting the yellow ball is 1/5.

Probability of Selecting Backoff Value (example)

IEEE 802.11 (WiFi) involves a station selecting a random backoff from 0 to 15. The probability of selecting 5 is 1/16.

Total Expectation (definition)

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:

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

Total Expectation of Packet Delay (example)

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.

Total Expectation of Password Length (example)

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.

Number of Attempts (definition)

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: $n/2$

Number of Attempts in Choosing Number (example)

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.

Number of Attempts in Choosing Key (example)

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).

Contents

Birthday Paradox (definition)

Given n random numbers selected from the range 1 to d, the probability that at least two numbers are the same is:

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

Two People Have Same Birthday (example)

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

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

Group Size for Birthday Matching (example)

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.

Group Size for Hash Collision (example)

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

Number Theory

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
number.tex, r1963

Contents

Divides (definition)

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 $b|a$.

Divides (example)

3 divides 12, since $12 = 4 \times 3$. Also, 3 is a divisor of 12, or $3|12$.

Greatest Common Divisor (definition)

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

Greatest Common Divisor (example)

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

Relatively Prime (definition)

Two integers, a and b, are relatively prime if $\gcd(a, b) = 1$.

Relatively Prime (example)

$\gcd(7, 12) = 1$, 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.

Relatively Prime (exercise)

How many positive integers less than 10 are relatively prime with 10?

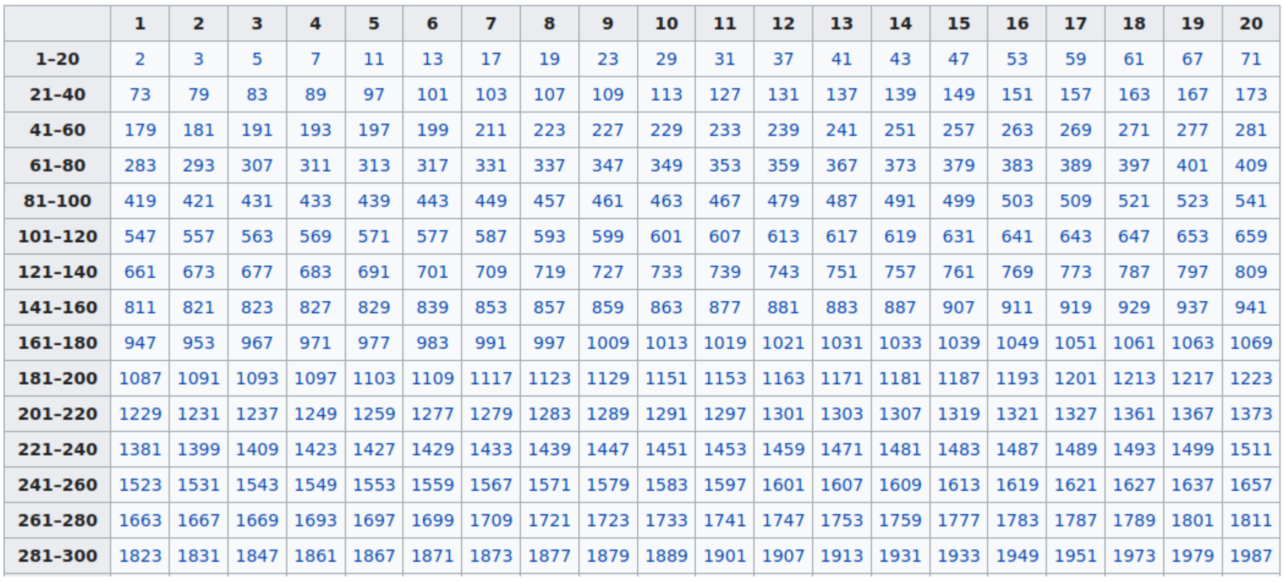
Prime Number (definition)

An integer $p > 1$ is a *prime number* if and only if its only divisors are $+1$, -1, $+p$ and $-p$.

Prime Number (example)

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.

First 300 Prime Numbers



	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

Prime Factors (definition)

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

Prime Factors (example)

The following are examples of integers expressed as prime factors:

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

Integers as Prime Factors (exercise)

Find the prime factors of 12870, 12936 and 30607.

Prime Factorization Problem (definition)

There are no known efficient, non-quantum algorithms that can find the prime factors of a sufficiently large number.

Prime Factorization Problem (example)

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

Euler's Totient Function (definition)

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 $\mathrm{Tot}(n)$.

Properties of Euler's Totient (definition)

Several useful properties of Euler's totient are:

$$\phi(1) = 1$$

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

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

Euler's Totient Function (example)

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

Contents

Modular arithmetic simple (definition)

Modular arithmetic is similar to normal arithmetic (addition, subtraction, multiplication, division) but the answers “wrap around”.

mod operator (definition)

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*.

mod operator (example)

The following are several examples of mod:

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

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

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

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

Congruent modulo (definition)

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 $\equiv b$.

Congruent modulo (example)

The following are examples of congruence:

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

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

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

Modular arithmetic (definition)

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

mod in Z_7 (example)

Consider the set:

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

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

Modular Arithmetic

- 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

Modular Addition (definition)

Addition in mod n is performed as normal addition, with the answer then mod by n.

Modular Addition (example)

The following are several examples of modular addition:

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

Additive Inverse (definition)

a is the *additive inverse* of b in mod n, if $a + b \equiv 0 \pmod{n}$.

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

Additive Inverse (example)

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:

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

Modular Subtraction (definition)

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.

Modular Subtraction (example)

For brevity, the modulus is sometimes omitted and $=$ is used in replace of $\equiv$. In mod 7:

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

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

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

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

Modular Multiplication (definition)

Modular multiplication is performed as normal multiplication, with the answer then mod by n.

Modular Multiplication (example)

In mod 7:

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

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

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

Multiplicative Inverse (definition)

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 $(\text{mod } n)$ if a is relatively prime to n.

Multiplicative Inverse in mod 7 (example)

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.

Multiplicative Inverse in mod 8 (example)

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.

Modular Division (definition)

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.

Modular Division (example)

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.

Properties of Modular Arithmetic (definition)

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

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

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

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

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

Contents

Fermat's Theorem 1 (definition)

If p is prime and a is a positive integer not divisible by p, then:

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

Fermat's Theorem 2 (definition)

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.

Fermat's theorem (example)

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.

Fermat's theorem (example)

What is $640^{163} \bmod 163$? Since 163 is prime, this matches Fermat's Theorem form 2. Therefore the answer is 640, or simplified to $640 \bmod 163 = 151$.

Euler's Theorem 1 (definition)

For every a and n that are relatively prime:

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

Euler's Theorem 2 (definition)

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.

Euler's theorem (example)

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.

Euler's theorem (example)

What is 13794^{4621} mod 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.

Contents

Modular Exponentiation (definition)

As exponentiation is just repeated multiplication, modular exponentiation is performed as normal exponentiation with the answer mod by n.

Modular Exponentiation (example)

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

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

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

Normal Logarithm (definition)

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.

Discrete Logarithm (definition)

If $b = a^i \pmod{p}$, then:

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

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

Primitive Root (definition)

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.

Primitive Root (example)

Consider the prime $p = 7$:

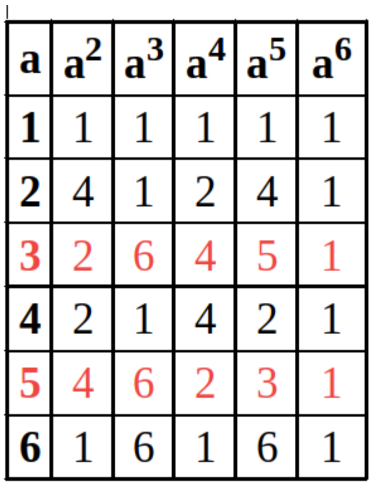
$a = 1 : 1^2 \bmod 7 = 1, 1^3 \bmod 7 = 1, \ldots$(not distinct)

$a = 2 : 2^2 \bmod 7 = 4, 2^3 \bmod 7 = 1, 2^4 \bmod 7 = 2, 2^5 \bmod 7 = 4, \ldots$(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).

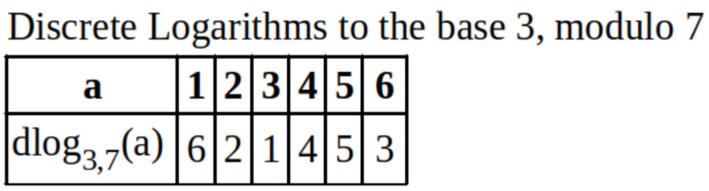
Powers of Integers, modulo 7



a	a^2	a^3	a^4	a^5	a^6
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

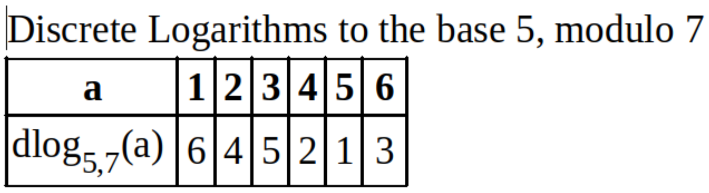
From the above table we see 3 and 5 are primitive roots of 7.

Discrete Logs, modulo 7



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

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.

Powers of Integers, modulo 17

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

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

Discrete Logarithms, modulo 17

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

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

Contents

Hard Problem: Integer Factorisation (definition)

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.

Hard Problem: Euler's Totient (definition)

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.

Hard Problem: Discrete Logarithms (definition)

Given b, a, and p, find i such that $i = \text{dlog}_{a,p}(b)$.

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.

Classical Ciphers

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
classical.tex, r1964

Contents

Caesar Cipher (algorithm)

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.

Caesar Cipher Encryption (exercise)

Using the Caesar cipher, encrypt plaintext `hello` with key 3.

How many keys are possible in the Caesar cipher? (question)

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?

Caesar Cipher Decryption (exercise)

You have received the ciphertext `TBBQOLR`. You know the Caesar cipher was used with key `n`. Find the plaintext.

Caesar Cipher, formal (algorithm)

$$C = E(K, P) = (P + K) \bmod 26 \quad (1)$$

$$P = D(K, C) = (C - K) \bmod 26 \quad (2)$$

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.

Caesar Cipher, formal (exercise)

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`.

Caesar Encrypt and Decrypt (python)

```
>>> 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 **??**).

Brute Force Attack (definition)

Try all combinations (of keys) until the correct plaintext/key is found.

Caesar Brute Force (exercise)

The ciphertext `FRUURJVBCANNC` was obtained using the Caesar cipher. Find the plaintext using a brute force attack.

Caesar Brute Force (python)

```
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.

Caesar Brute Force Results (text)

```
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).

How many attempts for Caesar brute force? (question)

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 $(26+1)/2 = 13.5$.

Recognisable Plaintext upon Decryption (assumption)

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.

Is plaintext always recognisable? (question)

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.

How to improve upon 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

Contents

Permutation (definition)

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.

Permutation (example)

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.

Monoalphabetic (Substitution) Cipher (definition)

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.

Monoalphabetic (Substitution) Cipher (example)

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`.

Decrypt Monoalphabetic Cipher (exercise)

Decrypt the ciphertext `QSWBSR` using the permutation chosen in the previous example.

How many keys in English monoalphabetic cipher? (question)

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.

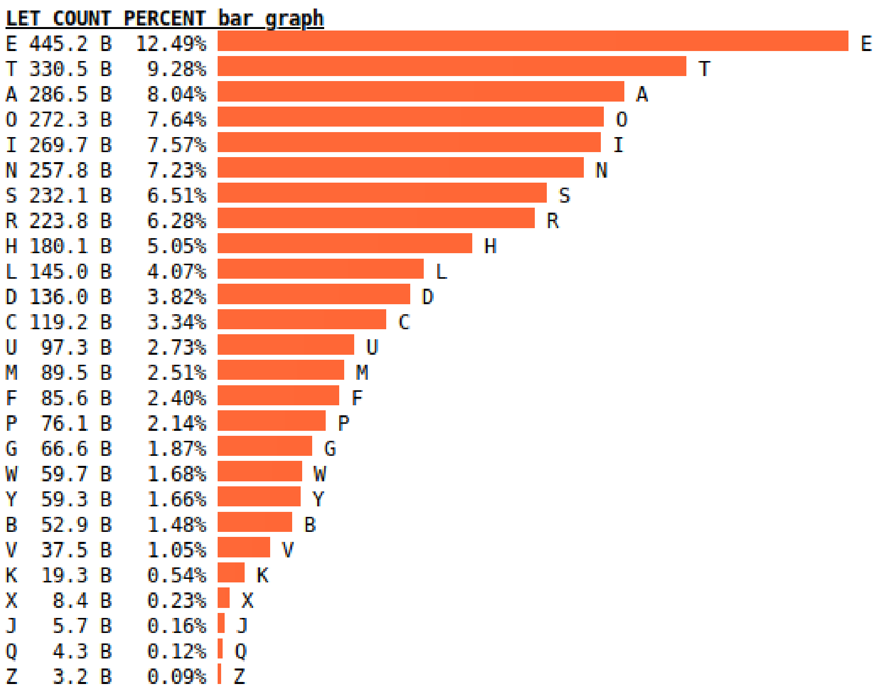
Brute Force on Monoalphabetic Cipher (exercise)

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.

Frequency Analysis Attack (definition)

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.

Relative Frequency of Letters by Norvig

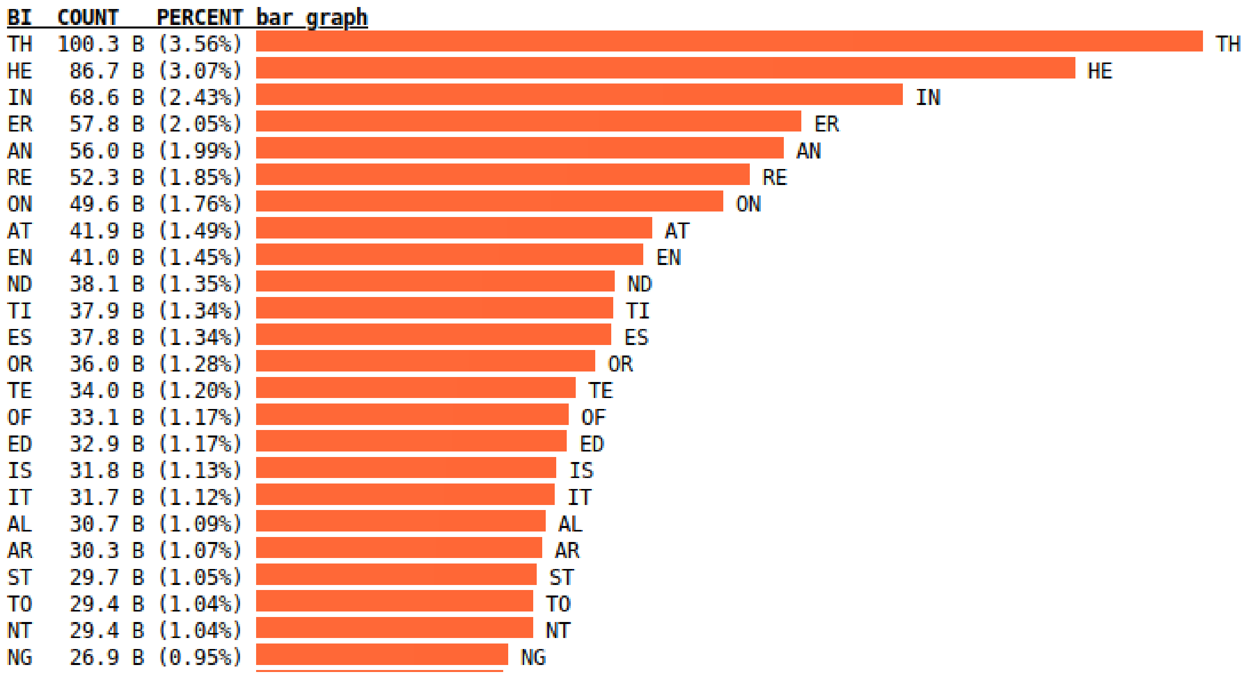


```
LET COUNT PERCENT bar graph
E 445.2 B  12.49%  E
T 330.5 B   9.28%  T
A 286.5 B   8.04%  A
O 272.3 B   7.64%  O
I 269.7 B   7.57%  I
N 257.8 B   7.23%  N
S 232.1 B   6.51%  S
R 223.8 B   6.28%  R
H 180.1 B   5.05%  H
L 145.0 B   4.07%  L
D 136.0 B   3.82%  D
C 119.2 B   3.34%  C
U  97.3 B   2.73%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

Credit: *Letter Counts* by Peter 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.

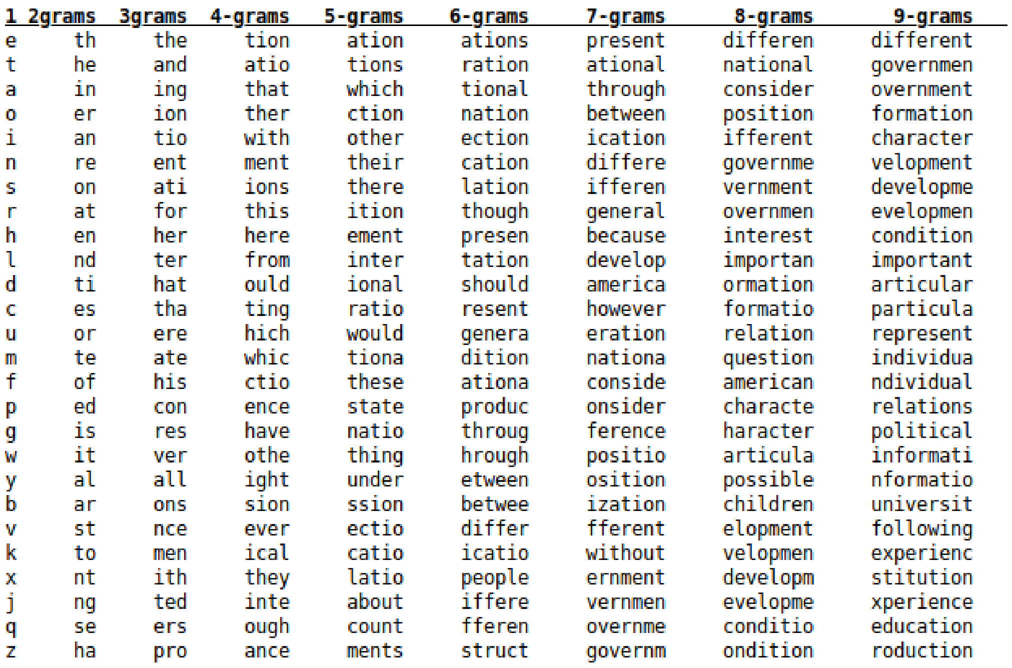
Relative Frequency of Digrams by Norvig



```
BI  COUNT    PERCENT bar graph
TH  100.3 B (3.56%)
HE   86.7 B (3.07%)
IN   68.6 B (2.43%)
ER   57.8 B (2.05%)
AN   56.0 B (1.99%)
RE   52.3 B (1.85%)
ON   49.6 B (1.76%)
AT   41.9 B (1.49%)
EN   41.0 B (1.45%)
ND   38.1 B (1.35%)
TI   37.9 B (1.34%)
ES   37.8 B (1.34%)
OR   36.0 B (1.28%)
TE   34.0 B (1.20%)
OF   33.1 B (1.17%)
ED   32.9 B (1.17%)
IS   31.8 B (1.13%)
IT   31.7 B (1.12%)
AL   30.7 B (1.09%)
AR   30.3 B (1.07%)
ST   29.7 B (1.05%)
TO   29.4 B (1.04%)
NT   29.4 B (1.04%)
NG   26.9 B (0.95%)
```

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

Relative Frequency of N-Grams 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

Break a Monoalphabetic Cipher (exercise)

Ciphertext:

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

Contents

Playfair Matrix Construction (algorithm)

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).

Playfair Matrix Construction (exercise)

Construct the Playfair matrix using keyword `australia`.

Playfair Encryption (algorithm)

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`.

Playfair Encryption (exercise)

Find the ciphertext if the Playfair cipher is used with keyword `australia` and plaintext `hello`.

Does Playfair cipher always map a letter to the same ciphertext letter? (question)

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?

Contents

Polyalphabetic (Substitution) Cipher (definition)

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.

Examples of Polyalphabetic Ciphers

- 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.

Contents

Vigenère Cipher (algorithm)

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.

Vigenère Cipher Encryption (example)

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`.

Vigenère Cipher Encryption (exercise)

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`.

Weakness of Vigenère Cipher

- 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.

Breaking Vigenère Cipher (example)

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.

Contents

Vernam Cipher (algorithm)

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.

XOR (python)

```
>>> 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.

Vernam Cipher Encryption (exercise)

Using the Vernam cipher, encrypt the plaintext 0111010101010000110110001 with the key 01011.

Vernam Cipher Encryption (python)

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

Contents

One-Time Pad (algorithm)

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 OTP if the keys are chosen appropriately.

Properties of OTP

- 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

Attacking OTP (example)

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.

Summary of 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.

Contents

Transposition vs Substitution

- 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

Rail Fence Cipher Encryption (definition)

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.

Rail Fence Encryption (exercise)

Consider the plaintext `securityandcryptography` with key 4. Using the rail fence cipher, find the ciphertext.

Rows Columns Cipher Encryption (definition)

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.

Rows Columns Encryption (exercise)

Consider the plaintext `securityandcryptography` with key 315624. Using the rows columns cipher, find the ciphertext.

Rows Columns Multiple Encryption (example)

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.

Summary of Transposition and Substitution Ciphers

- 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

Encryption and Attacks

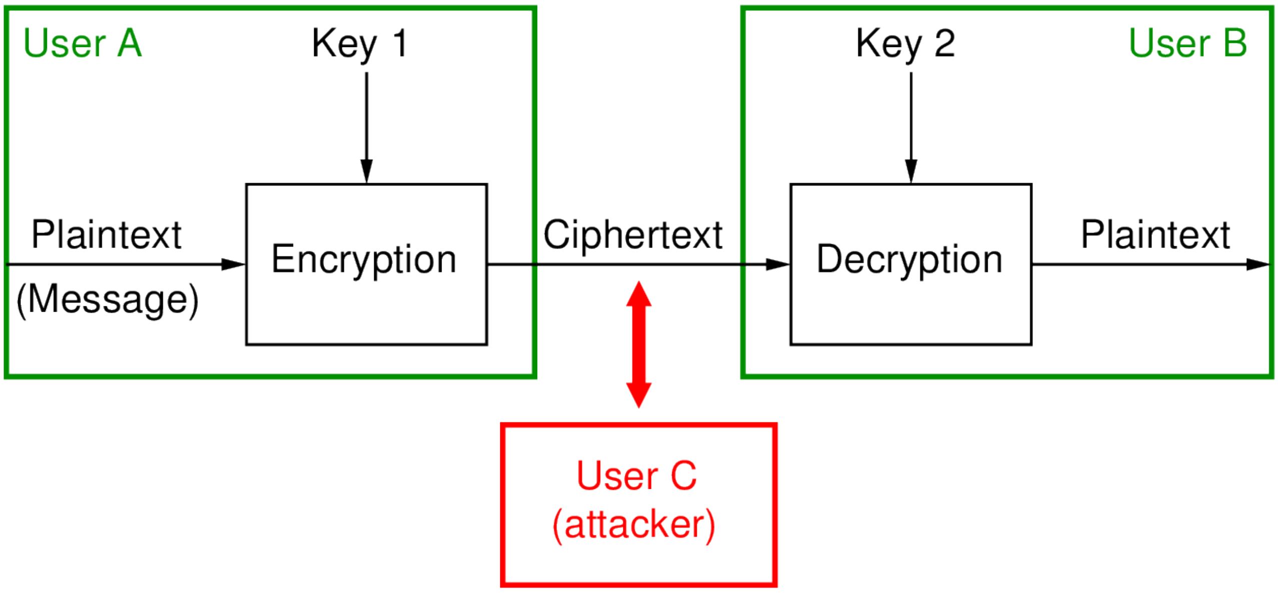
Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
encryption.tex, r1965

Contents

Model of Encryption for Confidentiality



The figure on slide 3 shows the general model for encrypting for confidentiality that we have seen previously.

Characterising Ciphers by Number of Keys

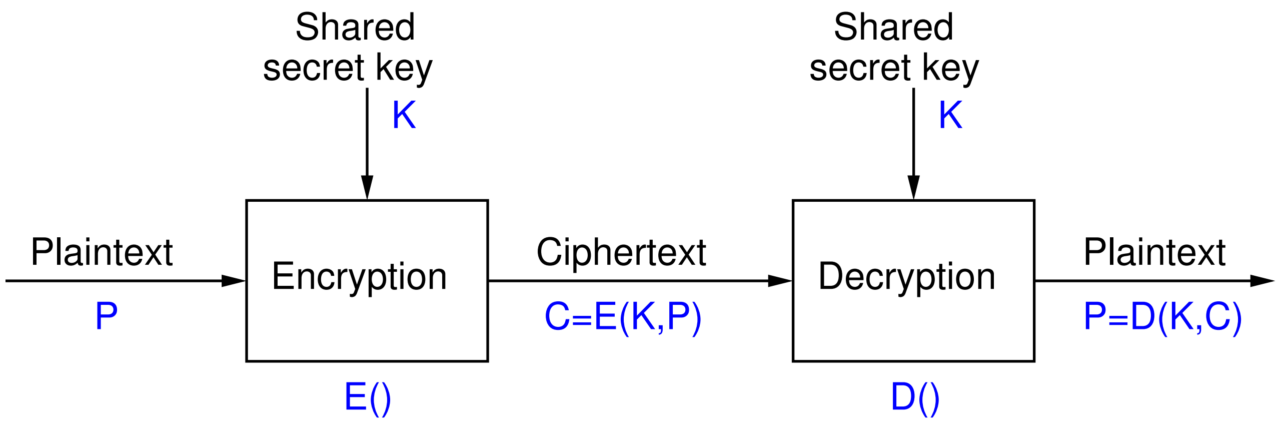
Symmetric	sender/receiver use same key (single-key, secret-key, shared-key, conventional)
Public-key	sender/receiver use different keys (asymmetric)

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.

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.

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.

Common Operations in Symmetric Ciphers

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.

Characterising Ciphers by Processing Plaintext

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.

Two Important Symmetric Key Block Ciphers

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

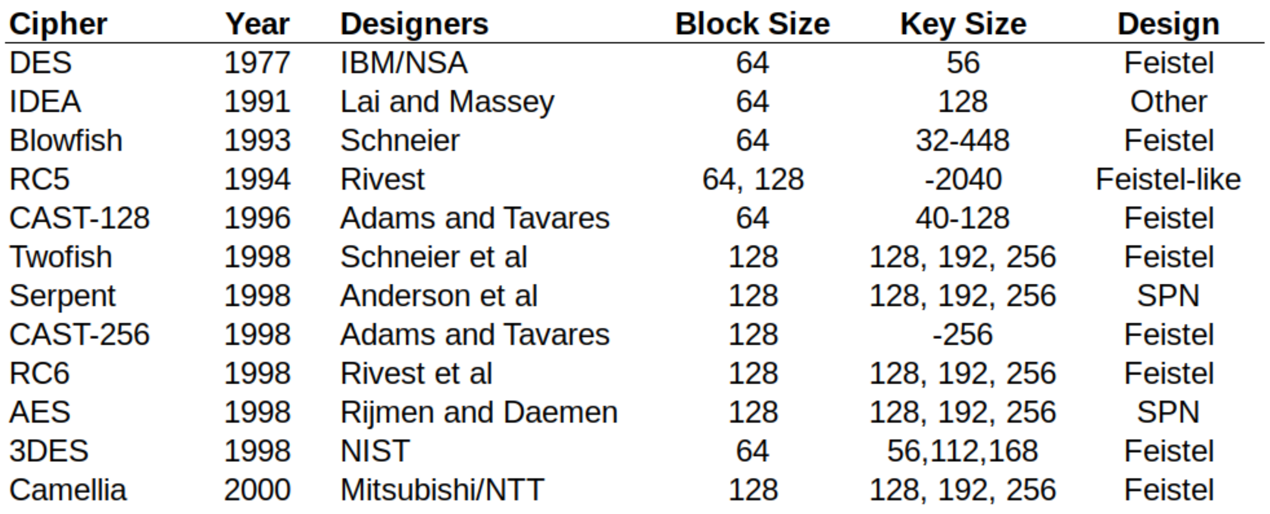
Advanced Encryption Standard (AES) Standardised a replacement of DES in 1998, and now widely used. Highly recommended for use.

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 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.

Common Symmetric Key Block Ciphers



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

The figure on slide 9 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

Contents

Aims and Knowledge of 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

Worst Case Brute Force Time for Different Keys

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

The table on slide 12 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.

Classifying Attacks Based Upon Information Known

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.

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 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 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 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.

General Measures of Security

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.

Common Metrics for Attacks

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.

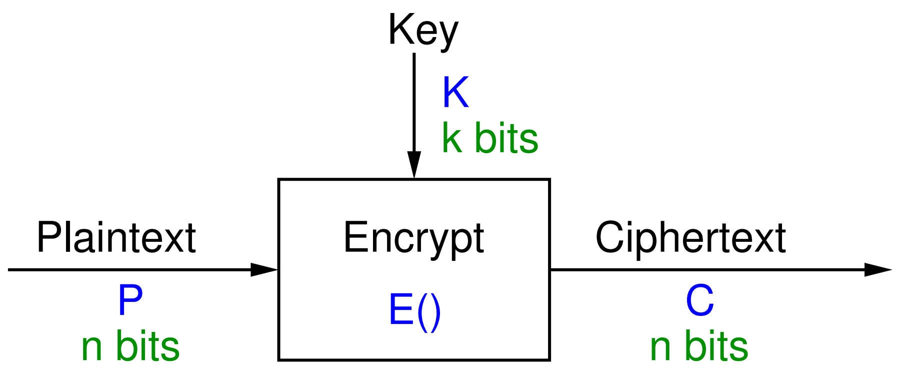
Often attacks requires intermediate values to be stored in memory while performing the attack. The less memory needed, the better the attack.

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.

Contents

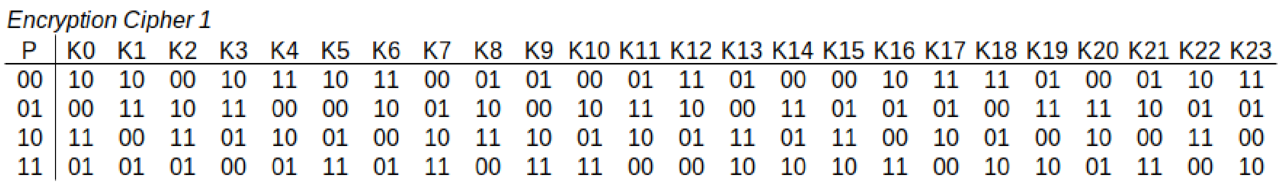
Block Cipher with n bit blocks

- 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



Modes of operation are covered in Chapter **??**.

Simple Ideal 2-bit Block Cipher 1


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

The figure on slide 22 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.

Encrypt with Ideal Cipher 1 (exercise)

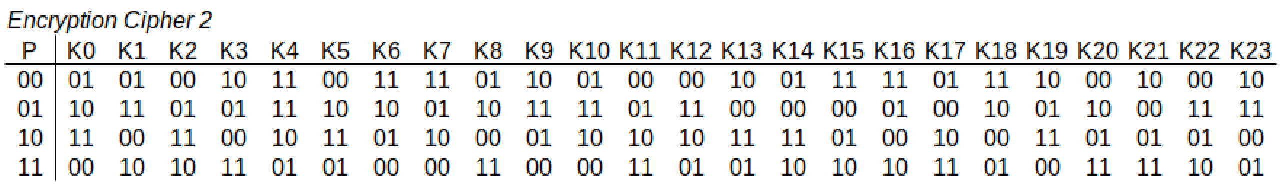
Encrypt the message *Tokyo* using the above ideal 2-bit block cipher 1 with key K6.

Issues When Applying Block Ciphers

- 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.

Simple Ideal 2-bit Block Cipher 2



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

The figure on slide 25 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.

What is plaintext with key K13, ciphertext 11 with ideal cipher 2? (question)

What is plaintext with key K13, ciphertext 11 with ideal cipher 2?

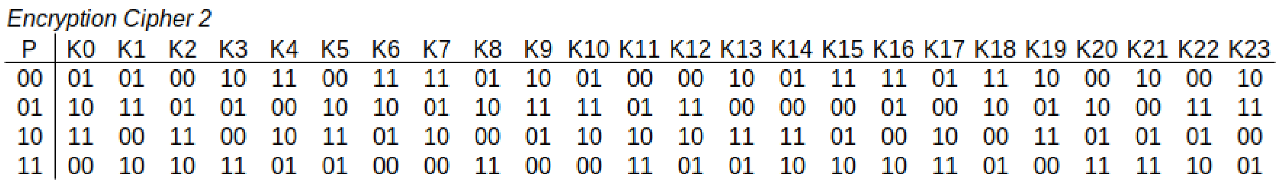
Decryption also involves a lookup. In the column for key `K13`, identify the ciphertext `11`, and the row indicates the original plaintext `10`.

What is plaintext with key K4, ciphertext 11 with ideal cipher 2? (question)

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.

Simple Ideal 2-bit Block Cipher 2 (fixed)



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

The figure on slide 28 shows the fixed cipher: it is now reversible, and decryption is possible for all values of key and ciphertext.

How many bits are needed to represent the key in cipher 2? (question)

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.

How to reduce repetition of plaintext blocks? (question)

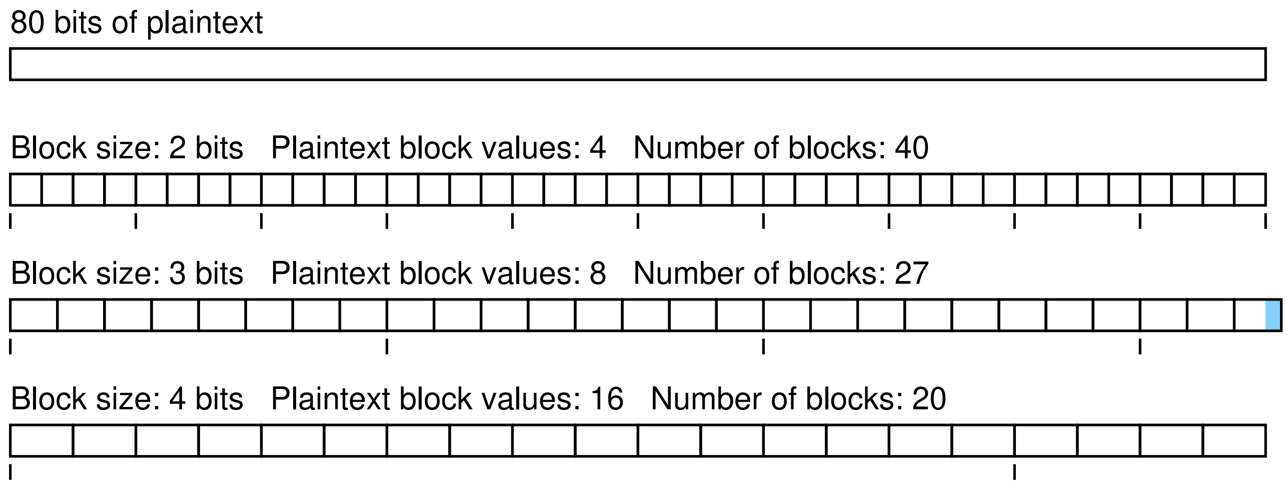
With a 2-bit ideal block cipher, with a long plaintext, many of plaintext blocks will repeat. This is bad for security (see Modes of Operation). What can you change in the design of an ideal block cipher that reduces repetition of plaintext blocks?

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.

Impact of Block Sizes for 80 bit Plaintext


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

The figure on slide 31 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.

General n-bit Ideal Block Cipher

- 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.

Ideal 64-bit Block Cipher (exercise)

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?

Feistel Structure for Block Ciphers

- 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

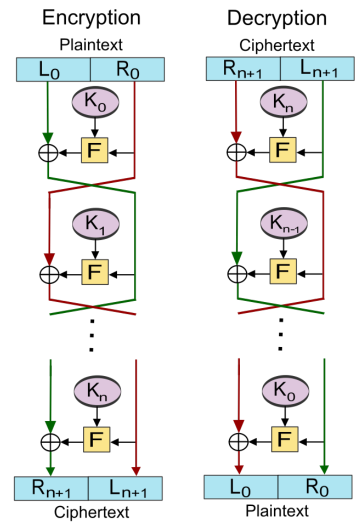
For example, with a 64-bit block cipher, there are 2^{64} possible mappings/keys, meaning the key length is $\log_2(2^{64}) = 64$ bits.

Diffusion and Confusion

- 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 **??**.

Feistel Encryption and Decryption



Credit: Amirki, `https://commons.wikimedia.org/wiki/File:Feistel_cipher_diagram_en.svg`, CC BY-SA 3.0

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.

Using the Feistel Structure

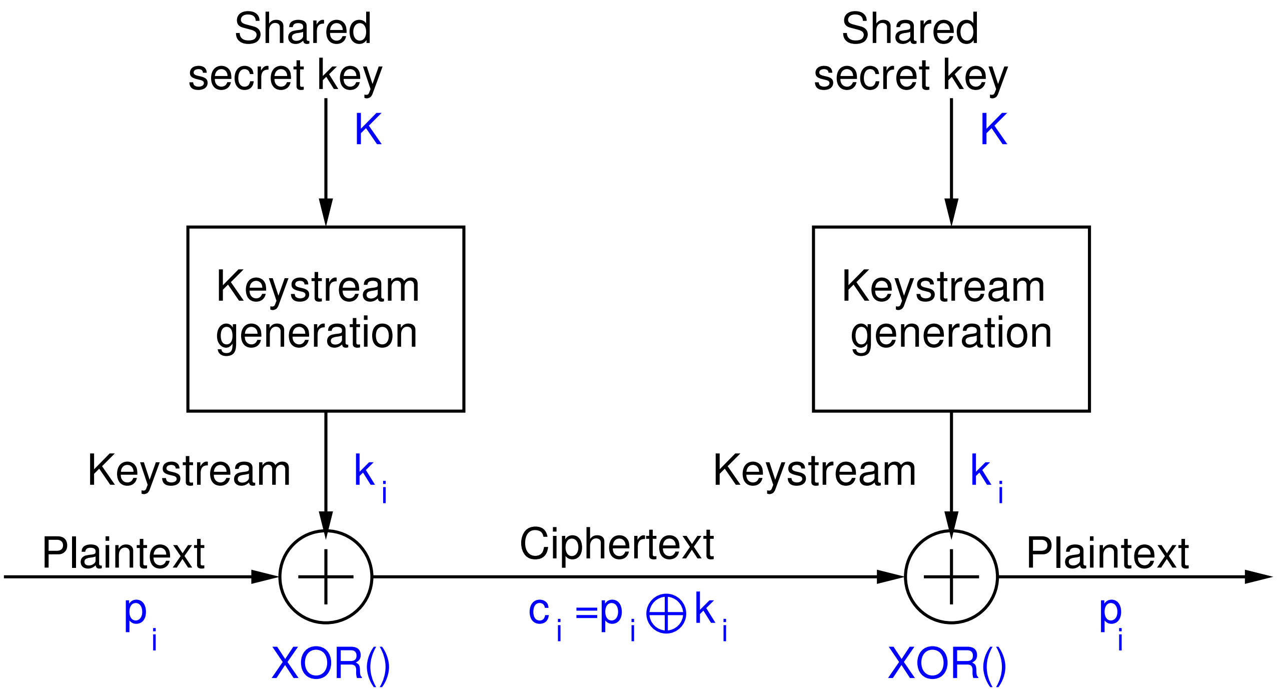
- Exact implementation depends on various design features
  - Block size, e.g. 64, 128 bits: larger values leads to more diffusion
  - Key size, e.g. 128 bits: larger values leads to more confusion, resistance against brute force
  - Number of rounds, e.g. 16 rounds
  - Subkey generation algorithm: should be complex
  - Round function F: should be complex
- Other factors include fast encryption in software and ease of analysis
- Trade-off: security vs performance

Contents

Stream Ciphers

- 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

Stream Cipher Encrypt and Decrypt



The figure on slide 40 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.

Key Re-use in Stream Ciphers

- 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

When can key re-use attack be successful if IV is used? (question)

If a stream cipher is using a *n*-bit 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.

Contents

DES and Real Brute Force Attacks

- 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

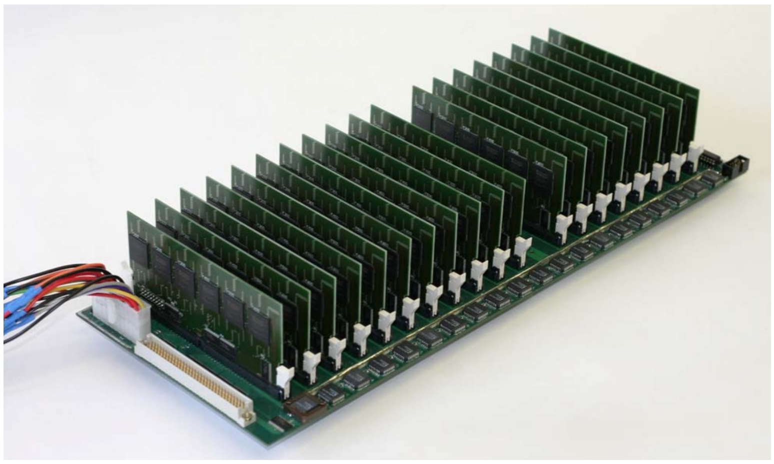
Paul Kocher and DeepCrack

- Developed by EFF
- Cost less than $US250,000
- 80×10^9 keys/sec
- Solved DES challenge in 56 hours
- See `www.cryptography.com` and `www.eff.org`

Credit: Wikimedia, CC0 1.0 Public Domain `https://commons.wikimedia.org/wiki/File:Paul_kocher_deepcrack.jpg`

COPACABANA by SciEngines, 2006

- Joint effort by SciEngines and German universities
- 120 FPGA, 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

Using the above example, we can roughly estimate what it would cost today to brute force DES.

Can We Estimate Cost Today?

- 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).

Contents

RIVYERA S3-5000 by SciEngines, 2013

- 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
  - 7.3×10^6 keys per mW

Credit: Copyright SciEngines GMBH

FPGA 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.

Breaking AES-128 in 2020

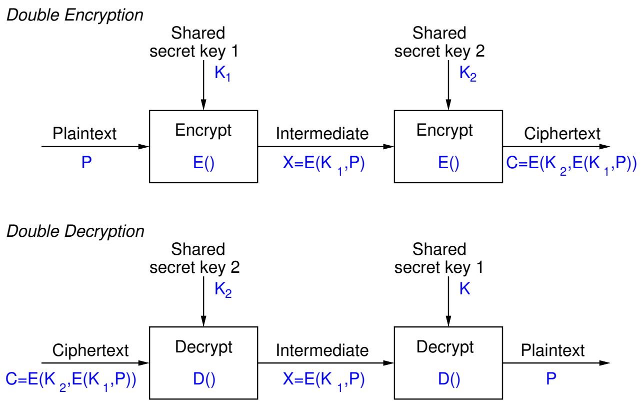
- AES-128 has key space of 2^{128}
- 2013: \$US12,800 for 5×10^8 k/s
- Assume: computers double speed every 1.5 years
- 2020: Increase by $2^5 = 32$; 1.6×10^{10} k/s
  - \$12,800: 6.7×10^{20} years
  - \$12,800,000: 6.7×10^{17} years
  - \$12,800,000,000: 6.7×10^{14} years
- Biclique attack about 2 to 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.

Contents

Double Encryption Concept

- Encrypt plaintext with one key, then encrypt output with another key



- 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

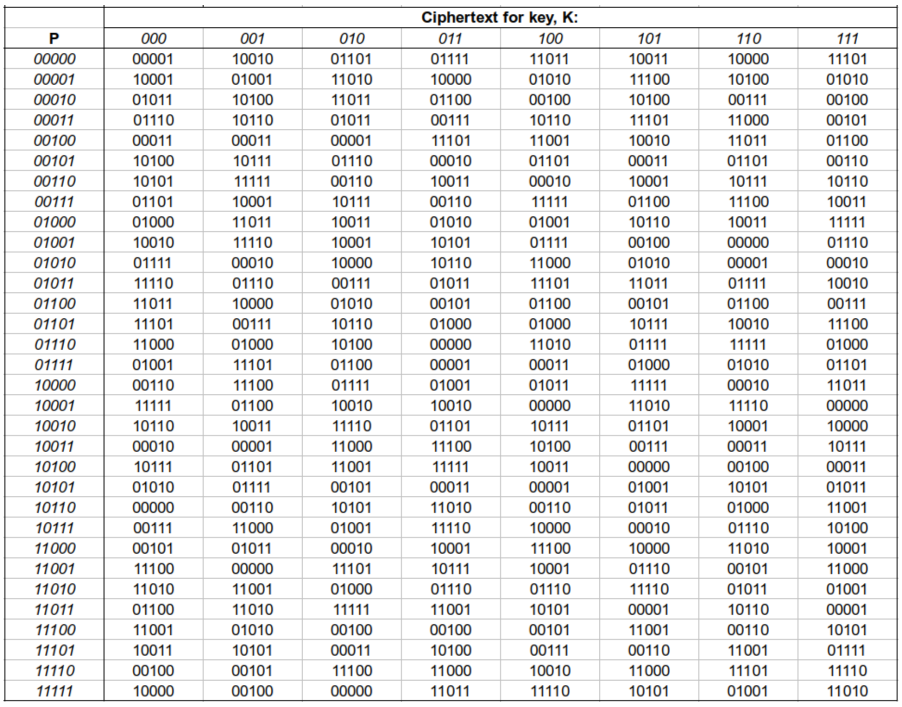
Double encryption was a (naive) option for extending the key length of DES. It effectively would double the key length from 56 bits to 112 bits. A new cipher would not have to be designed or analysed, and existing software/hardware implementations could be used.

But a meet-in-the-middle attack makes Double-DES (or double encryption on any block cipher) insecure.

Meet-in-the-Middle Attack

- 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)

Example 5-bit Block Cipher



	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

The figure on slide 54 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.

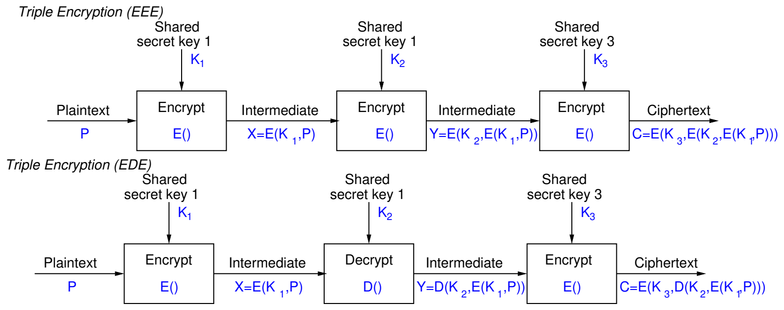
This example block cipher is used in the Meet-in-the-Middle attack exercise.

Meet-in-the-Middle Attack (exercise)

The figure on slide 54 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.

Triple Encryption Concept

- Different variations:
  - Use 2 keys, e.g. Triple-DES 112 bits
  - Use 3 keys, e.g. Triple-DES 168 bits



- 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

The figure on slide 56 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.

Contents

Cryptanalysis of Triple-DES and AES

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

Data Encryption Standard

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
des.tex, r1966

Contents

Data Encryption Standard

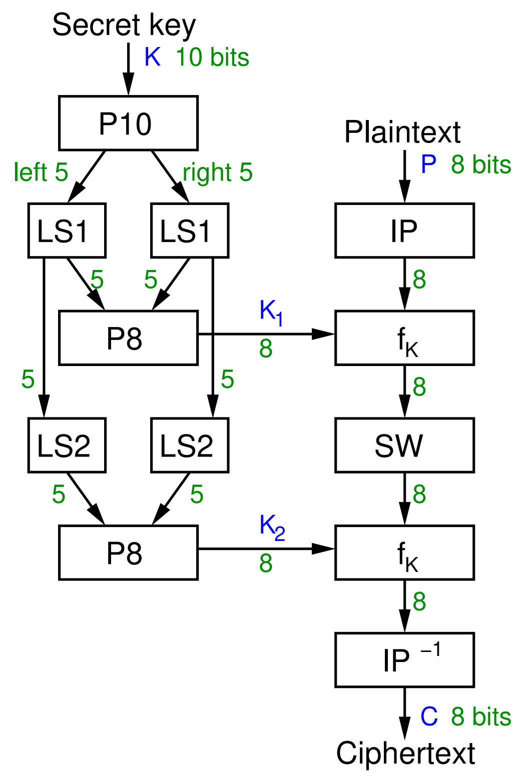
- Symmetric block cipher
- 56-bit key, 64-bit input block, 64-bit output block
- Developed in 1977 by NIST; designed by IBM (Lucifer) with input from NSA
- Principles used in other ciphers, e.g. 3DES, IDEA

Contents

Simplified DES

- Input (plaintext) block: 8-bits
- 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

S-DES Key Generation and Encryption

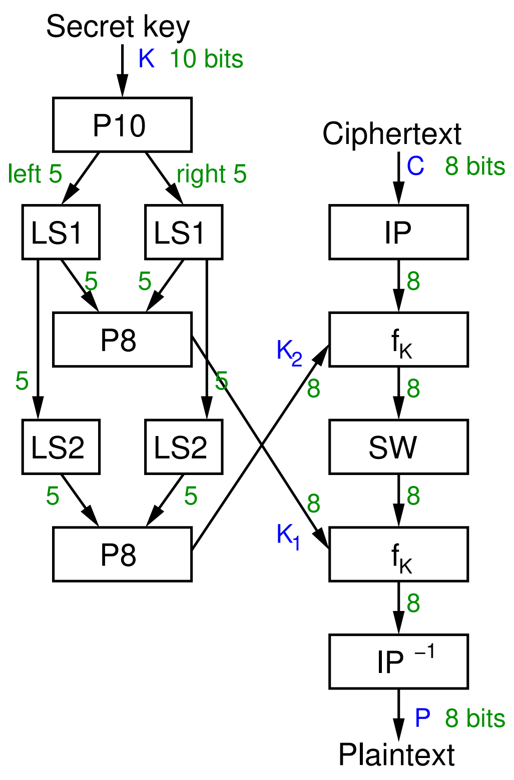


The figure on slide 6 shows the key generation and encryption steps of S-DES. Key generation, shown on the left, is used to generate round keys and is the same algorithm when used for both encryption and decryption. That is, the encrypter and decrypter will generate the exact same round keys.

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.

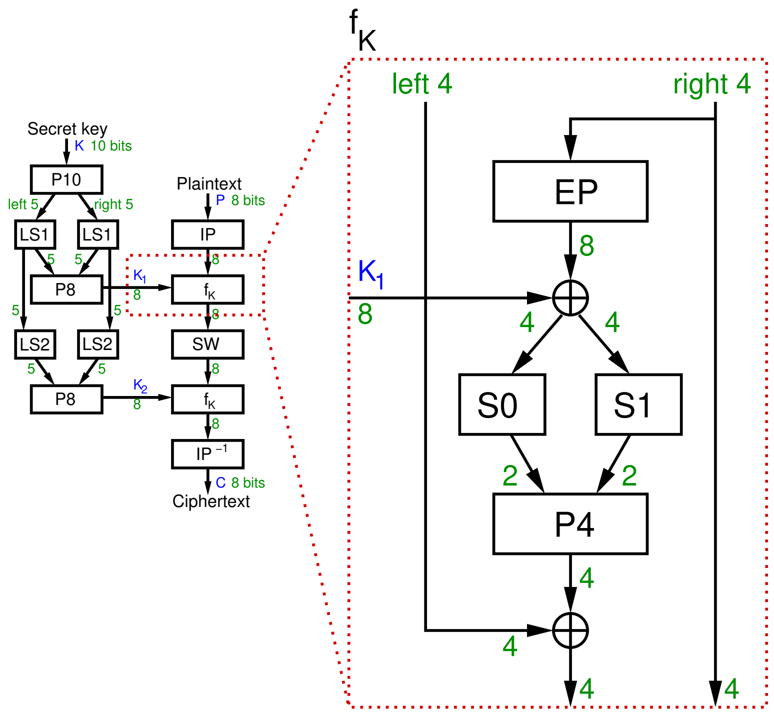
Encryption applies an Initial Permutation, then a round function f_k (with details to be shown shortly), SWaps the two halves of the 8 bit output, then reapplies the round function, but using the 2nd round key as input. Encryption ends with the inverse of the Initial Permutation.

S-DES Key Generation and Decryption



The figure on slide 7 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.

S-DES Round Function Details



The figure on slide 8 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.

S-DES Permutations (definition)

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.

Other Operations in S-DES

- LS-1: left shift by 1 position
- LS-2: left shift by 2 positions
- IP^{-1}: inverse of IP, such that $X = \text{IP}^{-1}(\text{IP}(X))$
- SW: swap the halves
- f_K: a round function using round key K
- F: internal function in each round

S-DES S-Boxes (definition)

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

$$S0 = \begin{bmatrix} 01 & 00 & 11 & 10 \\ 11 & 10 & 01 & 00 \\ 00 & 10 & 01 & 11 \\ 11 & 01 & 11 & 10 \end{bmatrix} \quad S1 = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 10 & 00 & 01 & 11 \\ 11 & 00 & 01 & 00 \\ 10 & 01 & 00 & 11 \end{bmatrix}$$

Encrypt with S-DES (exercise)

Show that when the plaintext 01110010 is encrypted using S-DES with key 1010000010 that the ciphertext obtained is 01110111.

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

1. Apply the initial permutation, IP, on P: 10101001

S-DES Summary

- 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*

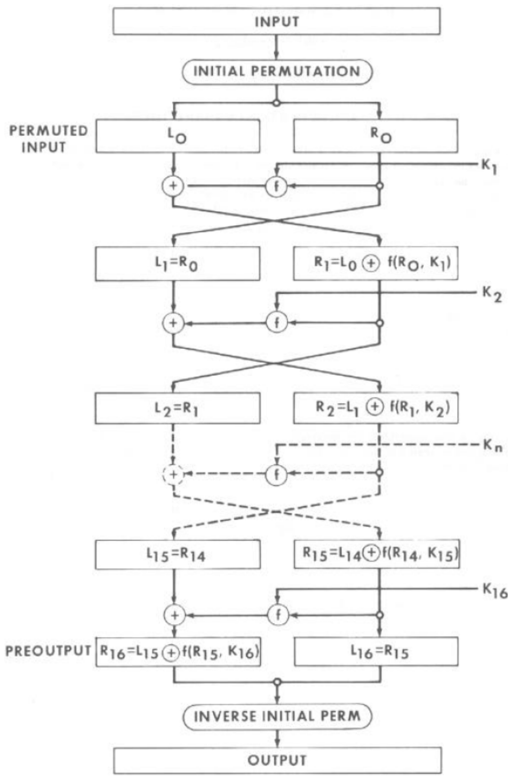
S-DES Compared to Real DES

- 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.

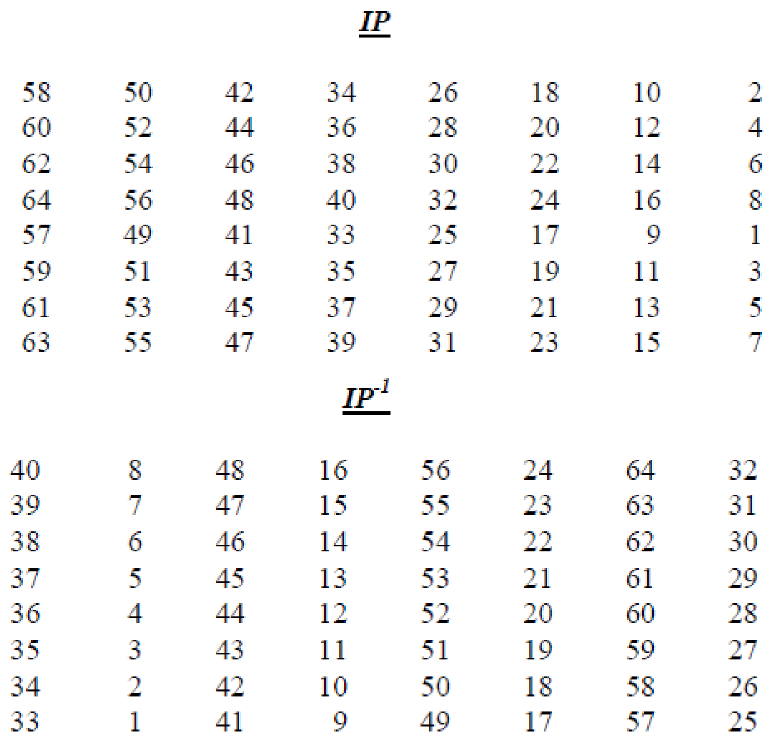
Contents

General DES Encryption Algorithm



The figure on slide 18 shows the overall steps in DES encryption. The details of each block are shown in the following.

Initial Permutation Tables for DES



IP

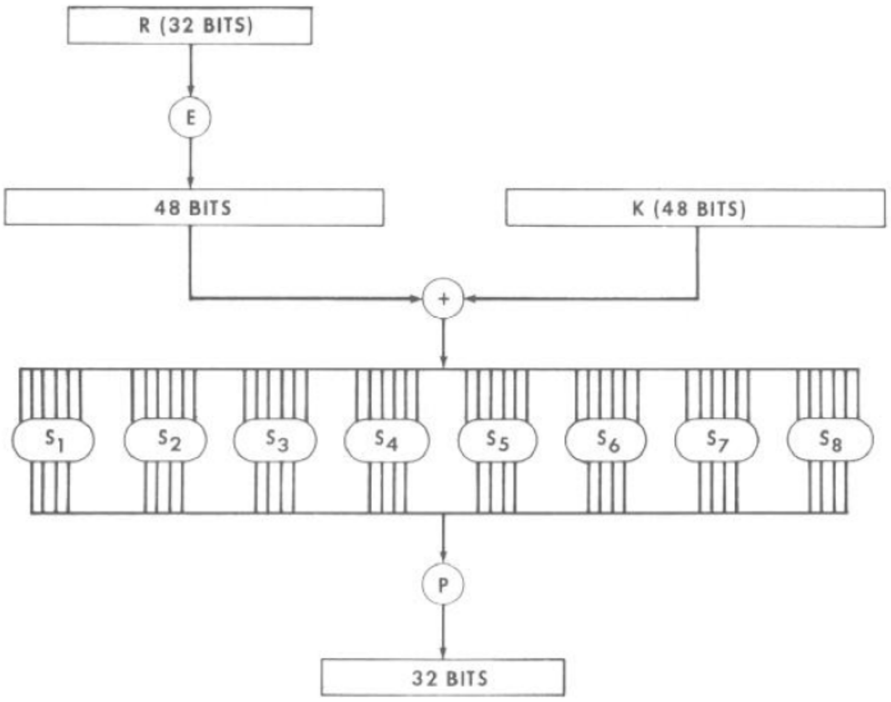
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

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

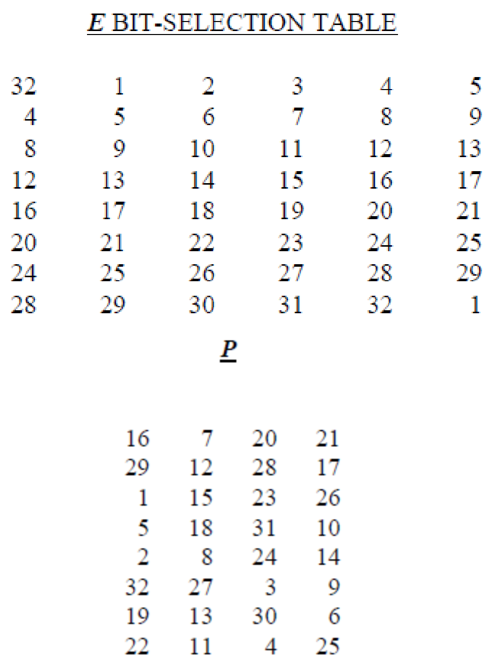
The figure on slide 19 shows the initial permutation and it's inverse. The table is read row-by-row. So the 58th input bit becomes the 1st output bit. The 50th input bit becomes the 2nd output bit. And the 7th input bit becomes the 64th output bit.

Calculation of F(R,K)



The figure on slide 20 shows the details of a single round of encryption, 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).

Permutation Tables for DES



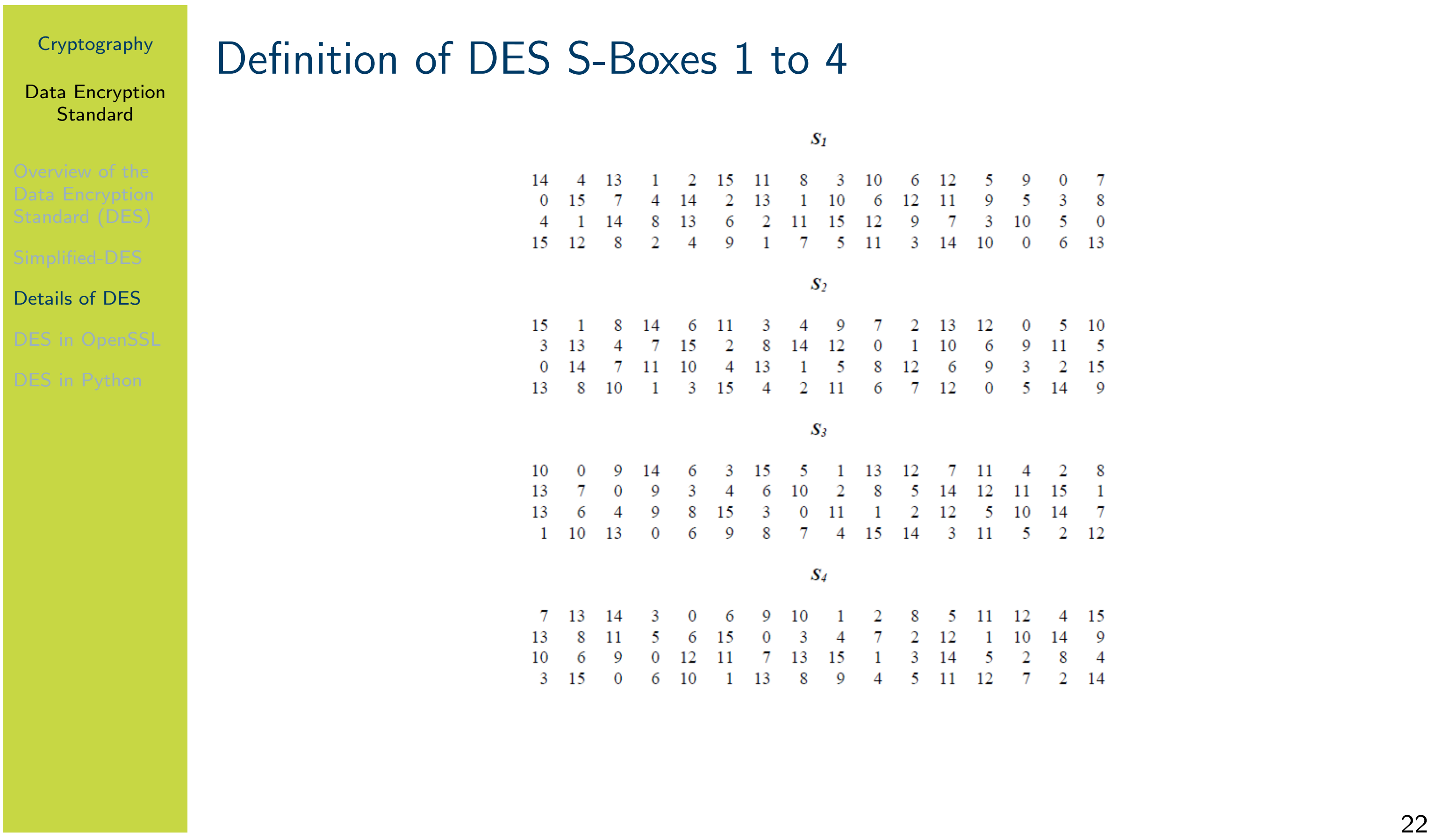
E **BIT-SELECTION TABLE**

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

The figure on slide 21 shows E and P which are used within a round of DES.



Definition of DES S-Boxes 1 to 4

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

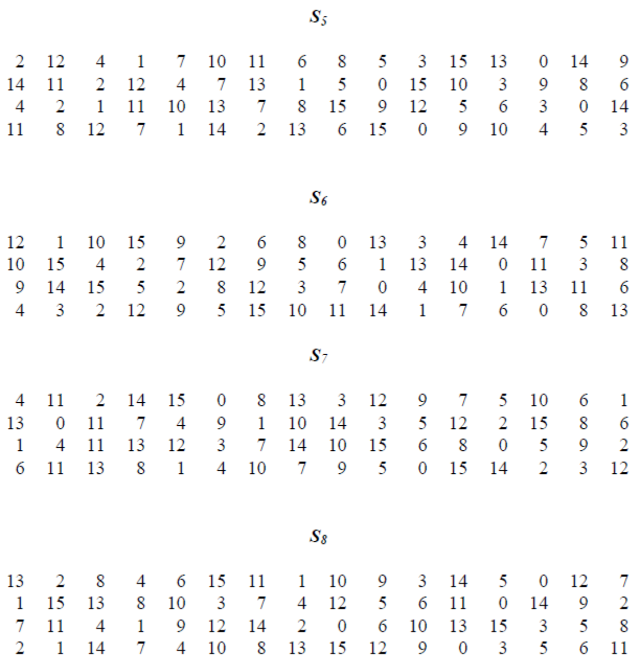
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

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

The figure on slide 22 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.

Definition of DES S-Boxes 5 to 6



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

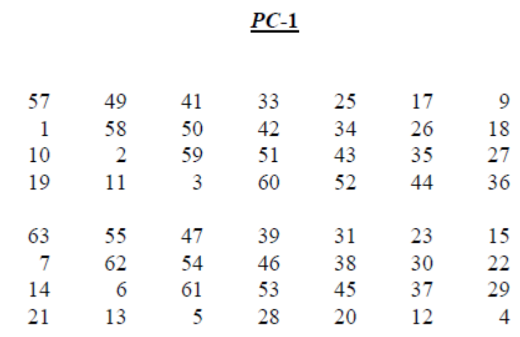
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

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

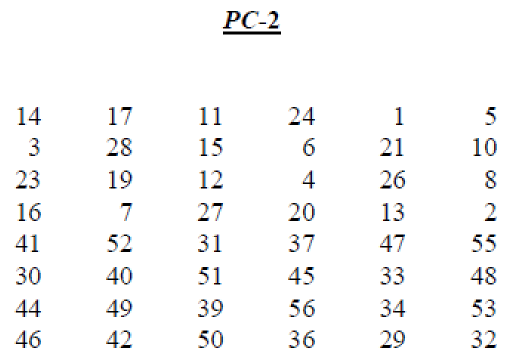
The figure on slide 23 shows the last 4 S-Boxes.

DES Permutated Choice 1 and 2



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

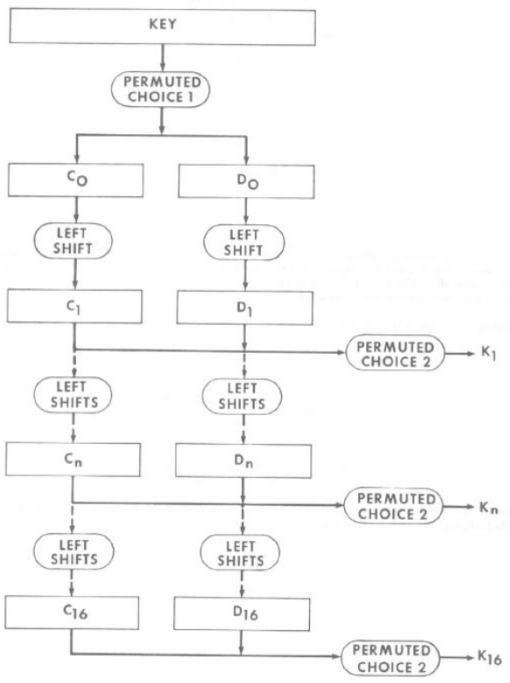


PC-2

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

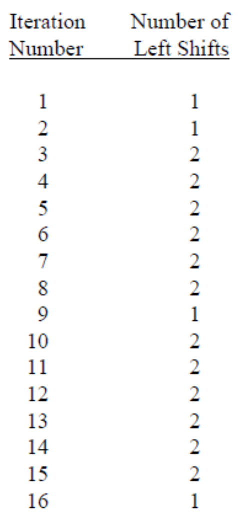
The figure on slide 24 shows the Permutated Choices used in key generation.

DES Key Generation Schedule



The figure on slide 25 shows the overall key generation steps.

DES Schedule of Left Shifts in Key Generation



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

The figure on slide 26 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.

Contents

DES Encryption in OpenSSL

- 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`

DES Key Generation (exercise)

Generate a shared secret key to be used with DES and share it with another person.

DES Encryption (exercise)

Create a message in a plain text file and after using DES, send the ciphertext to the person you shared the key with.

DES Decryption (exercise)

Decrypt the ciphertext you received.

Contents

AES in Python Cryptography Library

- cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/

Advanced Encryption Standard

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
aes.tex, r1967

Contents

History of AES

- 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

Selecting a Winner

- Original NIST criteria:
  - Security: effort to cryptoanalyse algorithm, randomness, . . .
  - 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

Selecting Rijndael for AES

- 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

Overview of AES

- 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
- 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

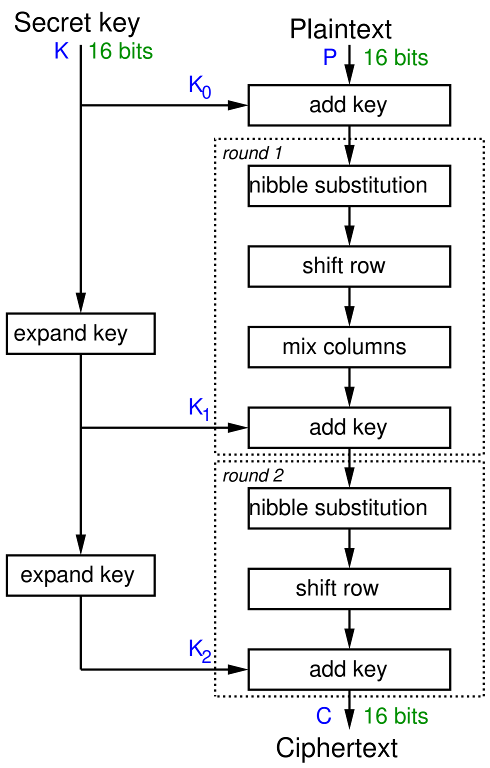
Contents

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.

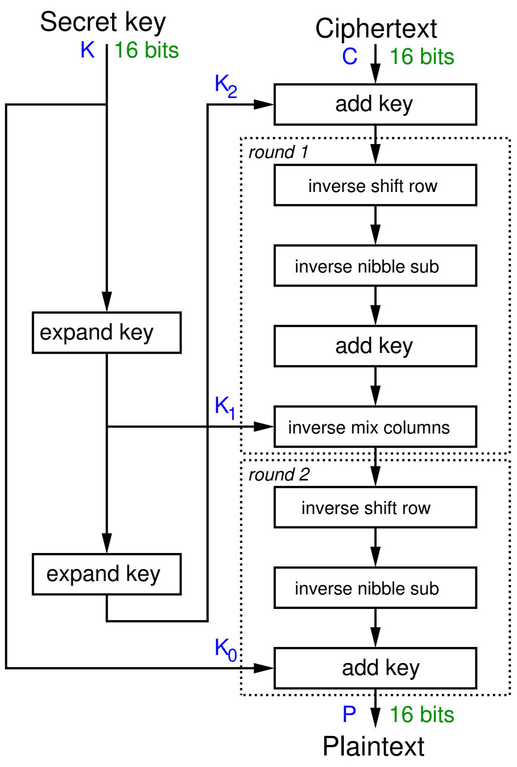
S-AES Encryption



The figure on slide 9 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. The figure on slide 11 shows that algorithm for K_1.

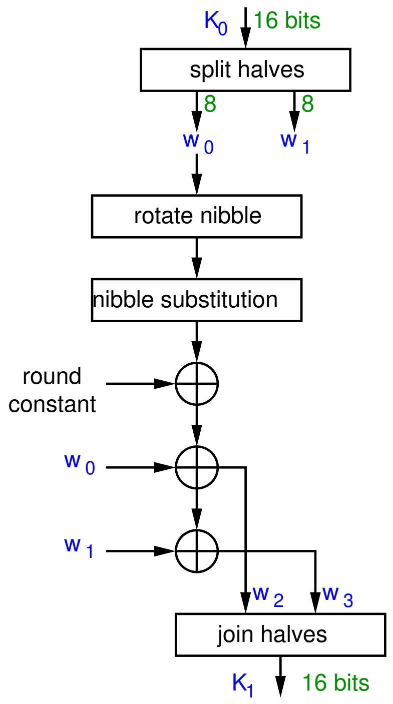
S-AES encryption operates on 16-bit blocks of plaintext. To encrypt, there is an initial *add key*, and then two rounds, where the 2nd round does not include the *mix columns* operation.

S-AES Decryption



The figure on slide 10 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.

S-AES Key Generation for Round 1



The figure on slide 11 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.

S-AES State Matrix (definition)

S-AES operates on a 16-bit state matrix, viewed as 4 nibbles

$$\begin{bmatrix} b_0 b_1 b_2 b_3 & b_8 b_9 b_{10} b_{11} \\ b_4 b_5 b_6 b_7 & b_{12} b_{13} b_{14} b_{15} \end{bmatrix} = \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix}$$

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.

S-AES Shift Row, Add Key and Rotate Nibbile operations (definition)

S-AES Shift Row:

$$\begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} \rightarrow \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,1} & S_{1,0} \end{bmatrix}$$

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.

S-AES S-Boxes (definition)

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.

S-AES Mix Columns (definition)

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.

S-AES Mix Columns (Simple) (definition)

Mix the columns in the state matrix be performing the following calculations.

Mix Columns:

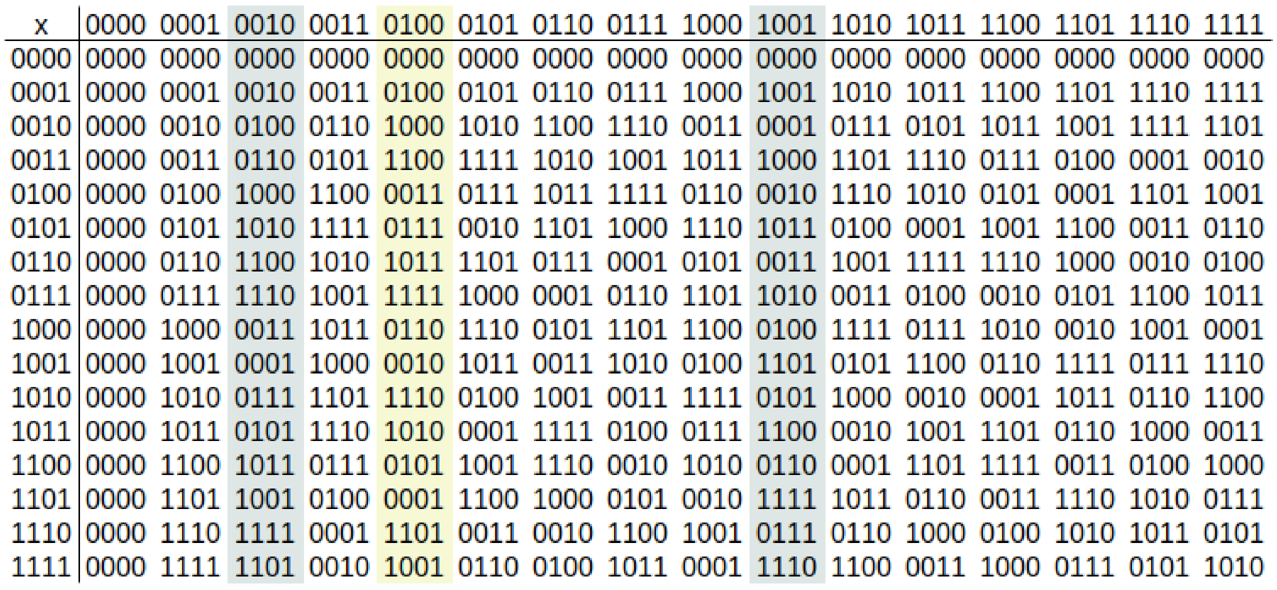
$$S'_{0,0} = S_{0,0} \oplus (0100 \times S_{1,0})$$
$$S'_{1,0} = (0100 \times S_{0,0}) \oplus S_{1,0}$$
$$S'_{0,1} = S_{0,1} \oplus (0100 \times S_{1,1})$$
$$S'_{1,1} = (0100 \times S_{0,1}) \oplus S_{1,1}$$

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 The figure on slide 17.

GF(2^4) Multiplication Table used in 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

The figure on slide 17 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).

Comparing S-AES and AES-128

- 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

Contents

Encrypt with S-AES (exercise)

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.

Contents

AES Key Generation (exercise)

Generate a shared secret key to be used with AES and share it with another person.

AES Encryption (exercise)

Create a message in a plain text file and after using AES, send the ciphertext to the person you shared the key with.

AES Decryption (exercise)

Decrypt the ciphertext you received.

AES Performance Benchmarking (exercise)

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).

Contents

AES in Python Cryptography Library

- `https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/`

Pseudorandom Number Generators

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 13 Feb 2020,
prng.tex, r1769

Block Cipher Modes of Operation

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 23 Dec 2021,
modes.tex, r1949

Contents

How Do Block Ciphers Encrypt Arbitrary Length Plaintext?

- 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)

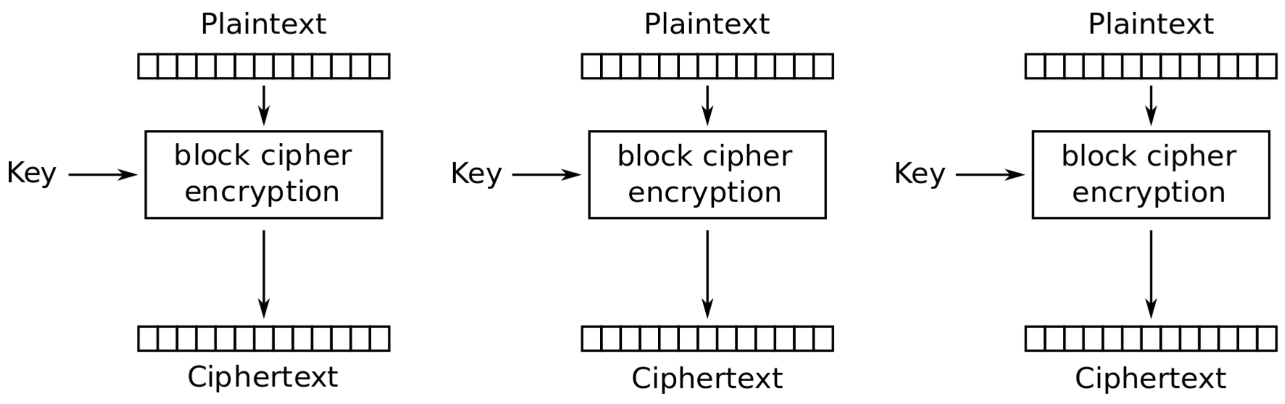
We will not cover each mode of operation in detail, but rather present them so you are aware of some of the common modes. For more technical details of some of these modes of operation, including discussion of padding, error propagation and the use of initialisation vectors, see NIST Special Publication 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.

Contents

ECB Summary

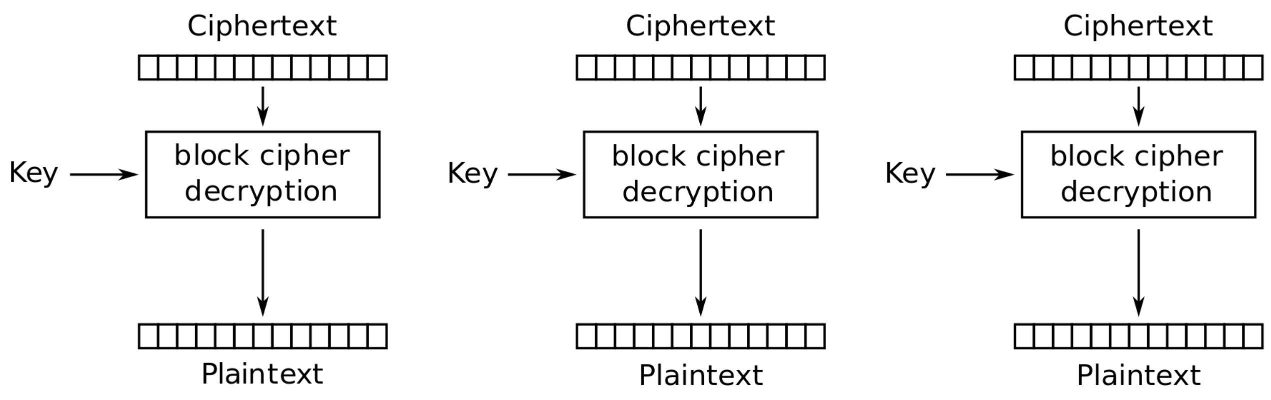
- Each block of 64 plaintext bits is encoded independently using same key
- Typical applications: secure transmission of single values (e.g. encryption key)
- Problem: with long message, repetition in plaintext may cause repetition in ciphertext

ECB Encryption



Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:ECB_encryption.svg`, public domain

ECB Decryption



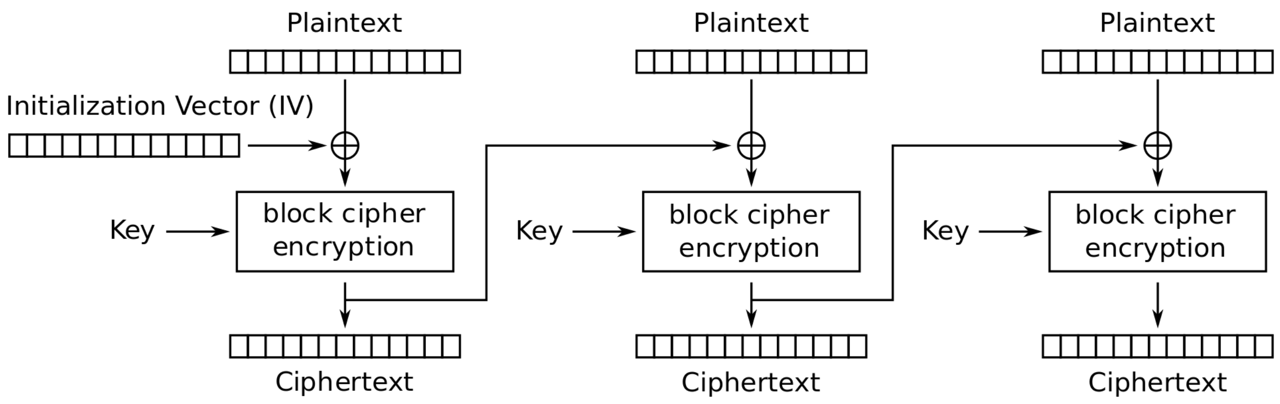
Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:ECB_decryption.svg`, public domain

Contents

CBC Summary

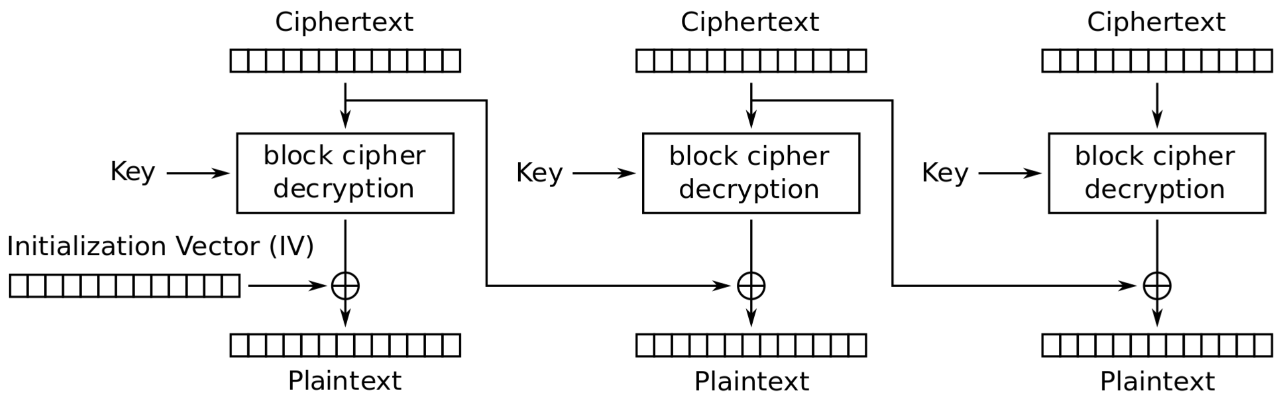
- Input to encryption algorithm is XOR of next 64-bits plaintext and preceding 64-bits ciphertext
- Typical applications: General-purpose block-oriented transmission; authentication
- Initialisation Vector (IV) must be known by sender/receiver, but secret from attacker

CBC Encryption



Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:CBC_encryption.svg`, public domain

CBC Decryption



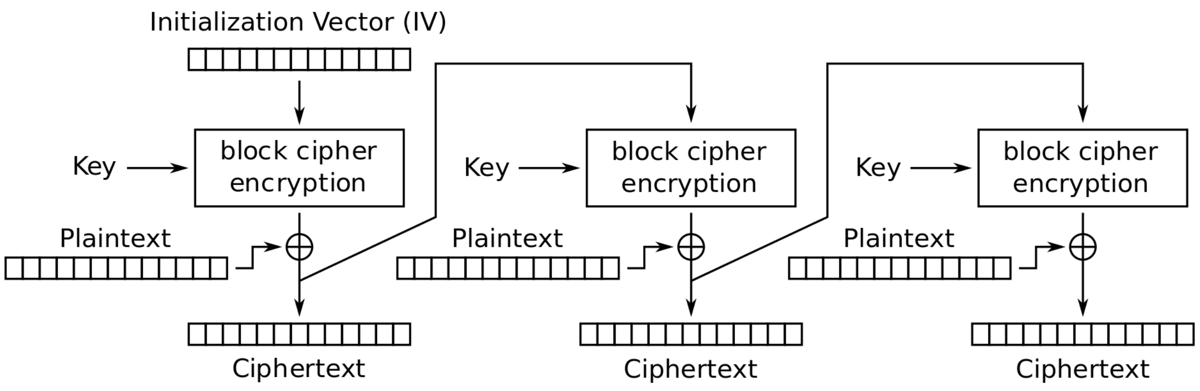
Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:CBC_decryption.svg`, public domain

Contents

CFB Summary

- 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

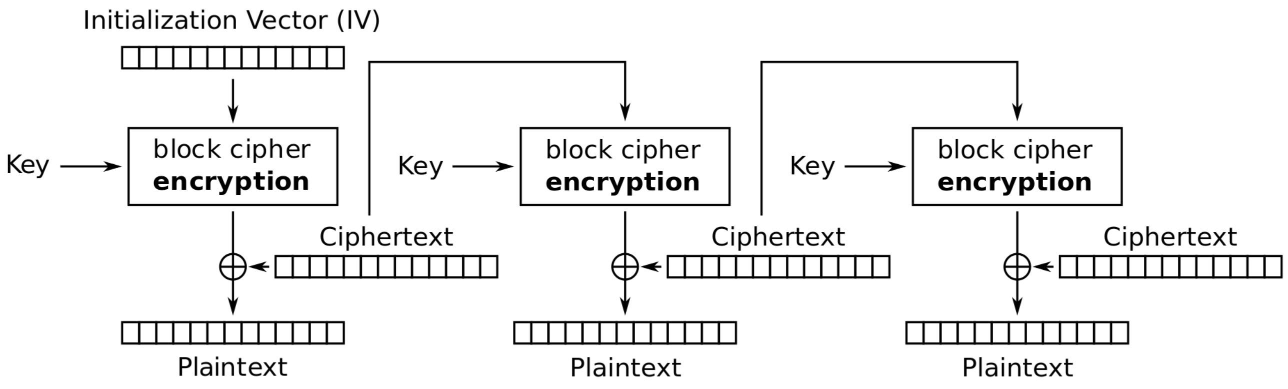
CFB Encryption



Cipher Feedback (CFB) mode encryption

Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CFB_encryption.svg, public domain

CFB Decryption



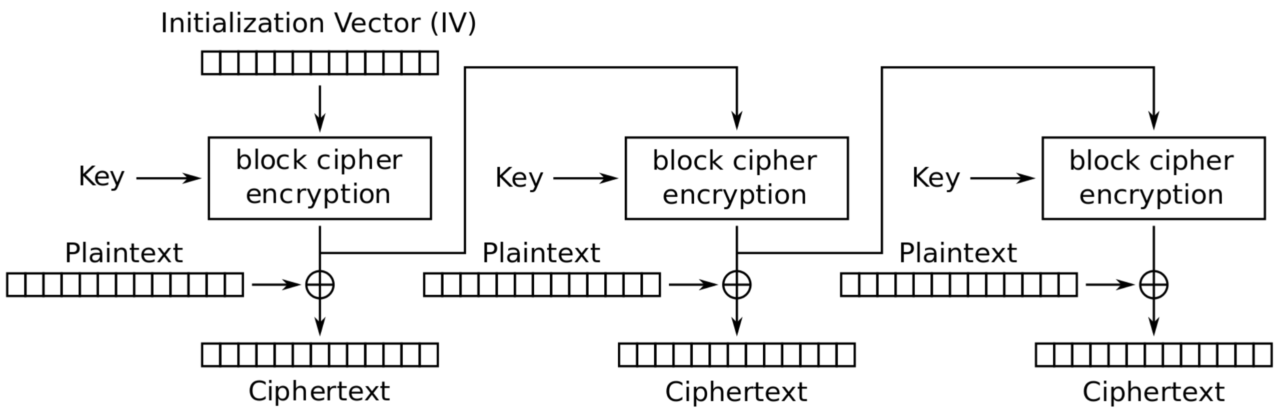
Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:CFB_decryption.svg`, public domain

Contents

OFB Summary

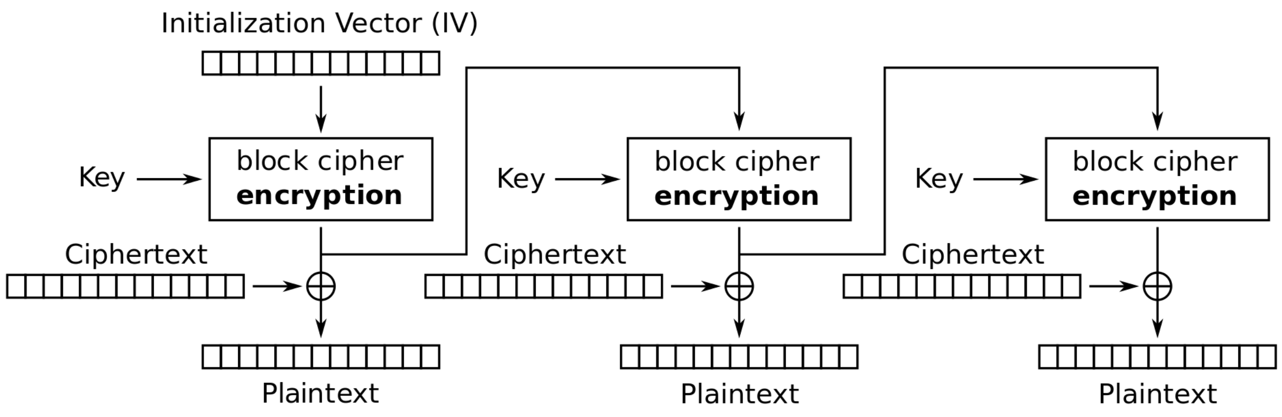
- Converts block cipher into stream cipher
- Similar to CFB, except input to encryption algorithm is preceding encryption output
- Typical applications: stream-oriented transmission over noisy channels (e.g. satellite communications)
- Advantage compared to OFB: bit errors do not propagate
- Disadvantage: more vulnerable to message stream modification attack

OFB Encryption



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:OFB_encryption.svg, public domain

OFB Decryption



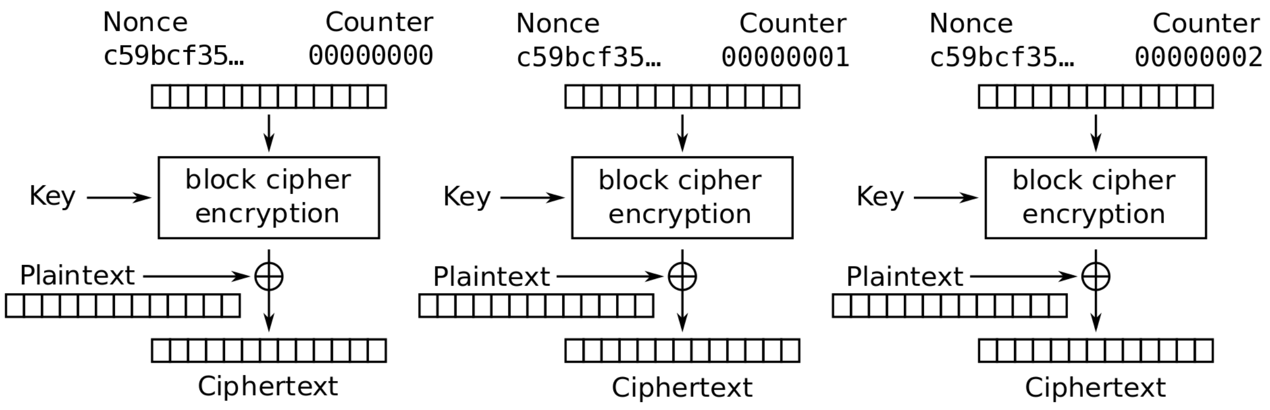
Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:OFB_decryption.svg`, public domain

Contents

CTR Summary

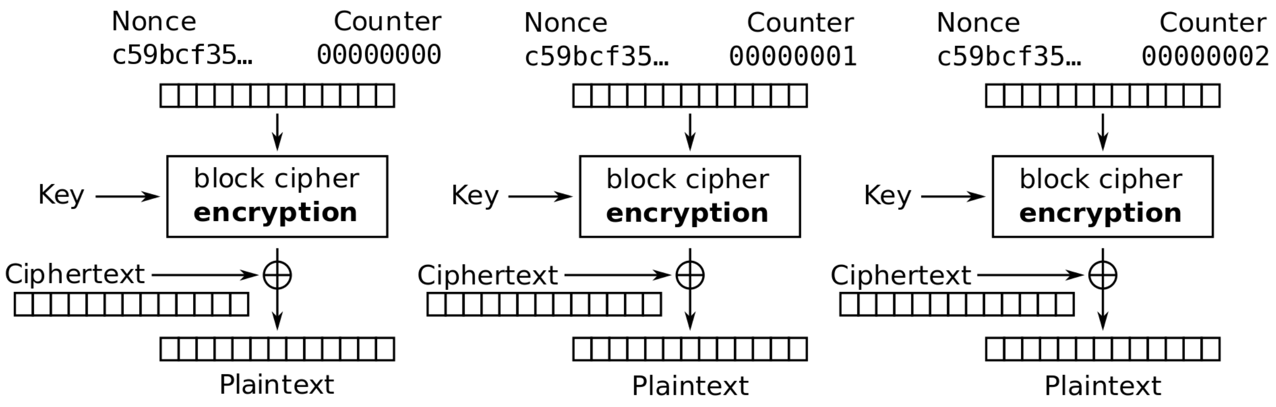
- Converts block cipher into stream cipher
- Each block of plaintext XORed with encrypted counter
- Typical applications: General-purpose block-oriented transmission; useful for high speed requirements
- Efficient hardware and software implementations
- Simple and secure

CTR Encryption



Credit: Wikimedia https://commons.wikimedia.org/wiki/File:CTR_encryption_2.svg, public domain

CTR Decryption



Credit: Wikimedia `https://commons.wikimedia.org/wiki/File:CTR_decryption_2.svg`, public domain

Contents

Encryption for Stored Data with XTS-AES

- 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

Public Key Cryptography

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 21 Dec 2021,
public.tex, r1944

Contents

Public Key vs Symmetric Key

- 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.

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.

Public and Private Keys

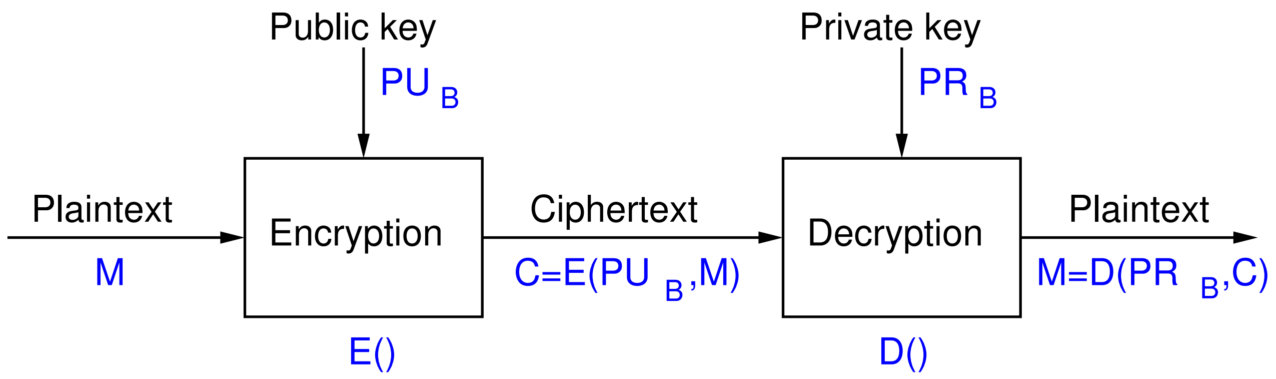
- Every user has their own key pair: (PU, PR)
  - Keys are generated using known algorithm (they are not chosen randomly like symmetric keys)
- Public key, PU
  - Available to everyone, e.g. in email signature, on website, in newspaper
- Private key, PR
  - Secret, known only by owner, e.g. access restricted 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.

Confidentiality with Public Key Crypto



- 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 $\rightarrow$ confidentiality

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.

Why Does Public Key Crypto Work?

- 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
  - Solving logarithms in modulo arithmetic
  - Solving logarithms on elliptic curves

The details of the algorithms are covered in subsequent chapters.

Public Key Crypto Examples

- 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

RSA

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 22 Dec 2021,
rsa.tex, r1945

Contents

RSA Public Key Algorithm

- 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 **??**.

The RSA Algorithm for Encryption

- 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.

RSA Key Generation (algorithm)

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.

RSA Key Generation (exercise)

Assume user A chose the primes $p = 17$ and $q = 11$. Find the public and private keys of user A.

RSA Encryption and Decryption (algorithm)

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.

Requirements of the RSA Algorithm

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.

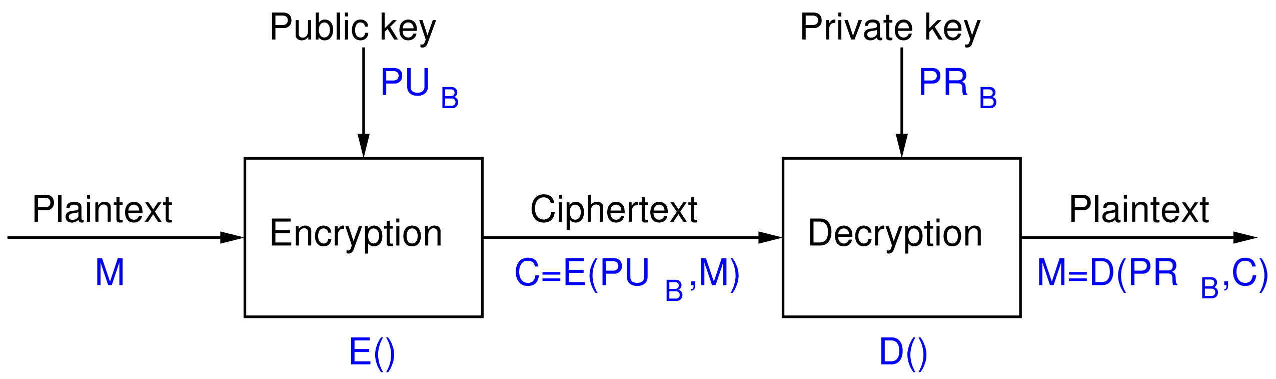
Ordering of RSA Keys

- 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.

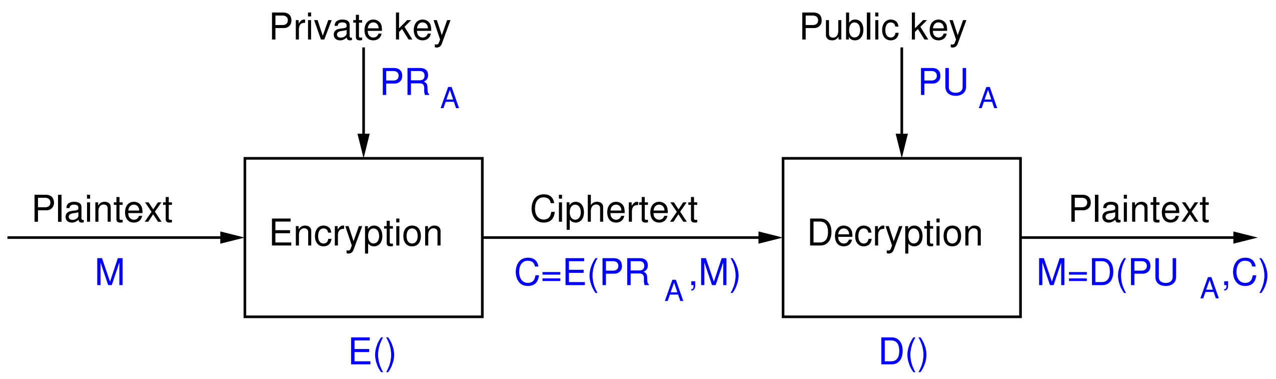
Why does authentication work? Since the sender is the only user that knows their private key, then they are the only user that can sign the message/plaintext. And the receiver can verify it came from that user if the signature decrypts successful with the sender's public key.

RSA used for Confidentiality



The figure on slide 10 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.

RSA used for Authentication



The figure on slide 11 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.

RSA Encryption for Confidentiality (exercise)

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.

RSA Decryption for Confidentiality (exercise)

Show that user A successfully decrypts the ciphertext.

Contents

Why Does RSA Decryption Work?

- 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 $\phi(n)$.

Parameter Selection 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).

Security of 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,

Progress in Factorisation

- 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.

Contents

Recommended or Typical RSA Parameters

- 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.

Decryption with Large d is Slow

- 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.

RSA Implementation Example

- 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).

Contents

RSA Key Generation (exercise)

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).

RSA Signing (exercise)

Create a message in a file, sign that message using the `dgst` command, and then send the message and signature to another person.

RSA Verification (exercise)

Verify the message you received.

RSA Performance Test (exercise)

Using the OpenSSL `speed` command, compare the performance of RSA encrypt/sign operation against the RSA decrypt/verify operation.

Contents

RSA in Python Cryptography Library

- `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/`

Diffie–Hellman Key Exchange

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 20 Feb 2020,
dh.tex, r1798

Contents

Diffie–Hellman Key Exchange

- 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).

If you do not know what a discrete logarithm is, it is worth refreshing your knowledge in number theory from Chapter **??**.

Diffie–Hellman Key Exchange (algorithm)

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 $\rightarrow$ 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 $\rightarrow$ 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.

Diffie–Hellman Key Exchange (exercise)

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.

Contents

Requirements 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.

Prove Identical Keys in DHKE (question)

Prove that user A and user B will always calculate the same shared secret key in DHKE. That is, prove that $K_A = K_B$.

Brute Force Attack on PR in DHKE (question)

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?

Discrete Logarithm Attack in DHKE (exercise)

Assuming a brute force attack is not possible, write an equation that the attacker would have to solve to find PR_A.

Discrete Logarithm is Computationally Hard Problem

- 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

Contents

MITM Attack on DHKE (exercise)

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.

Contents

Selecting Public Parameters p and g

- 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.

Contents

Contents

DHKE in Python Cryptography Library

- `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/`

Elliptic Curve Cryptography

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 23 Dec 2021,
elliptic.tex, r1949

Contents

Elliptic Curve (definition)

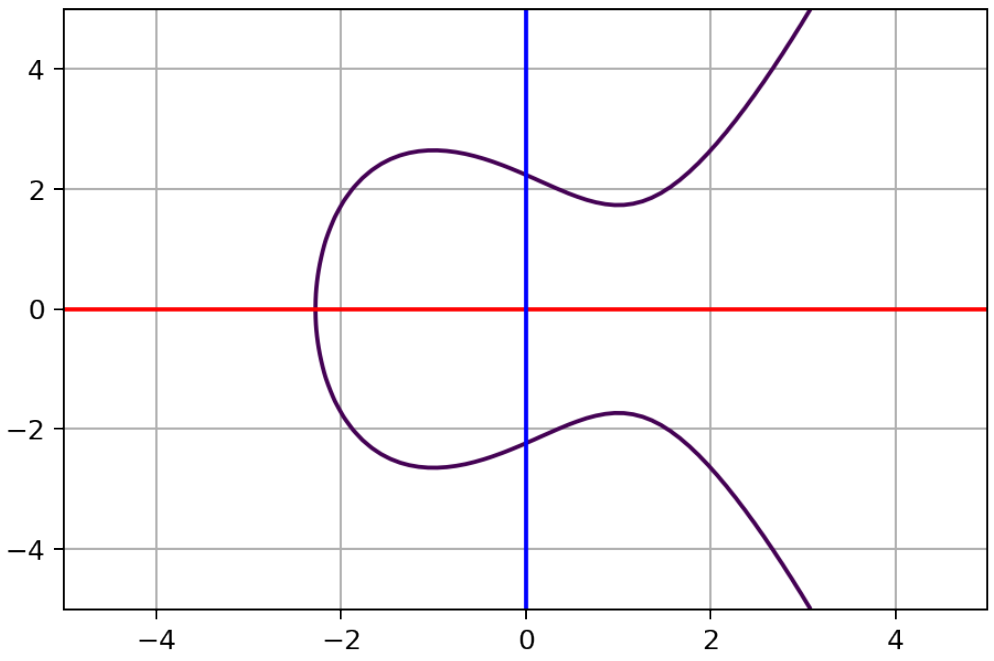
An elliptic curve is defined by:

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

(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.

Elliptic Curve for $y^2 = x^3 - 3x + 5$



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

The figure on slide 4 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.

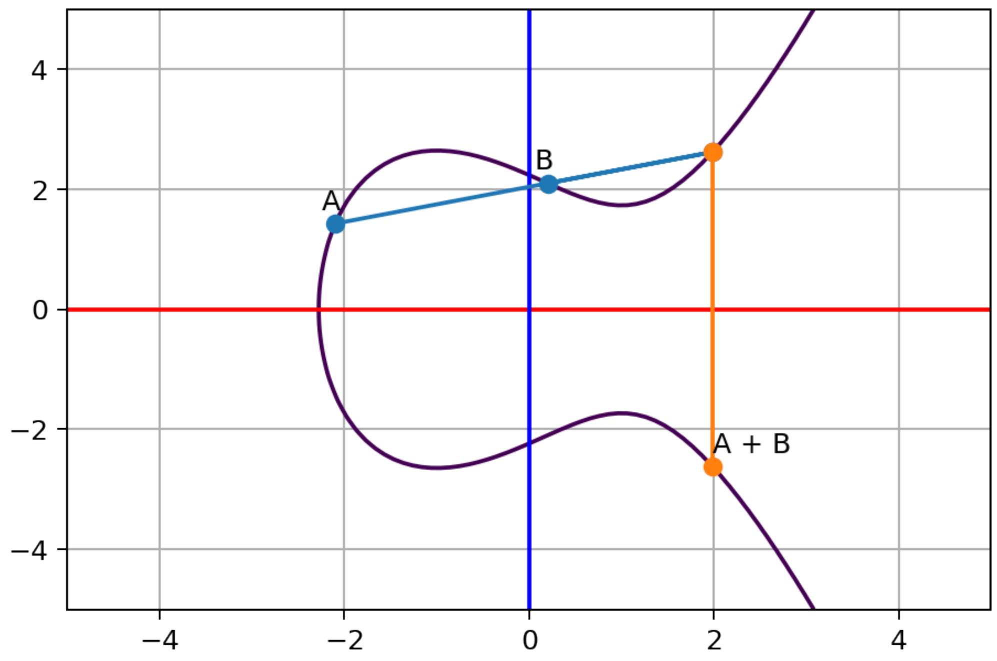
Addition Operation with an Elliptic Curve (definition)

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.

Addition Operation on Elliptic Curve

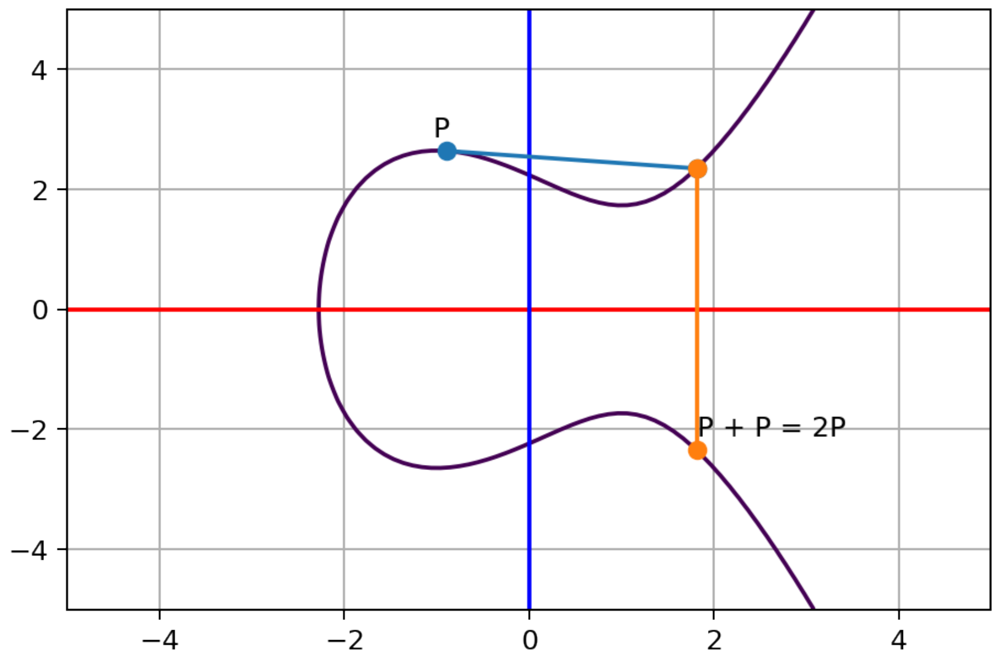


Credit: Generated based on MIT Licensed code by Fang-Pen Lin

The figure on slide 6 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.

Self Addition on Elliptic Curve

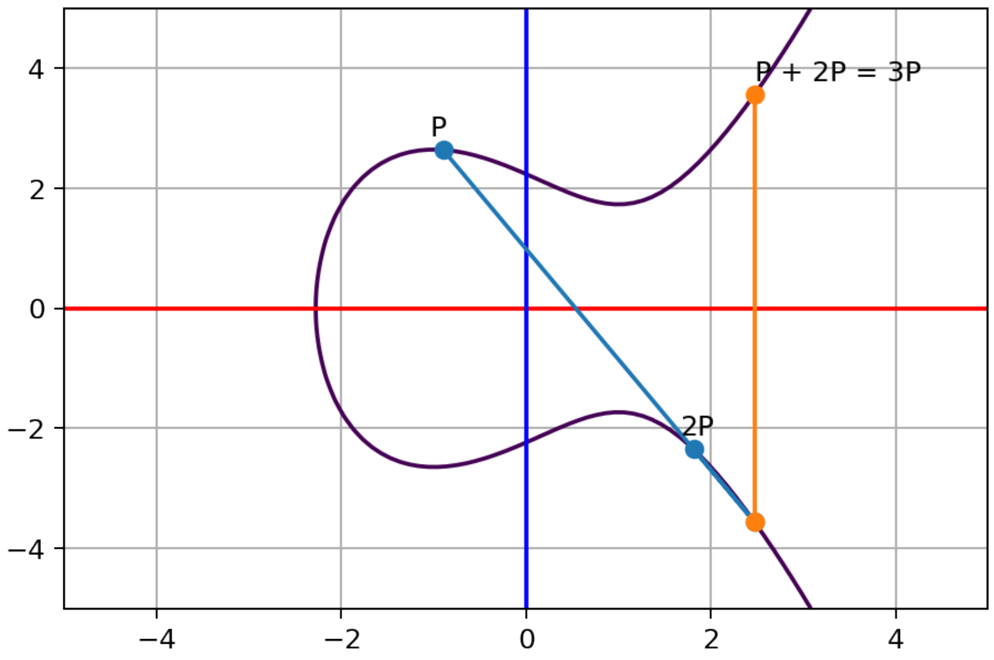


Credit: Generated based on MIT Licensed code by Fang-Pen Lin

The figure on slide 7 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.

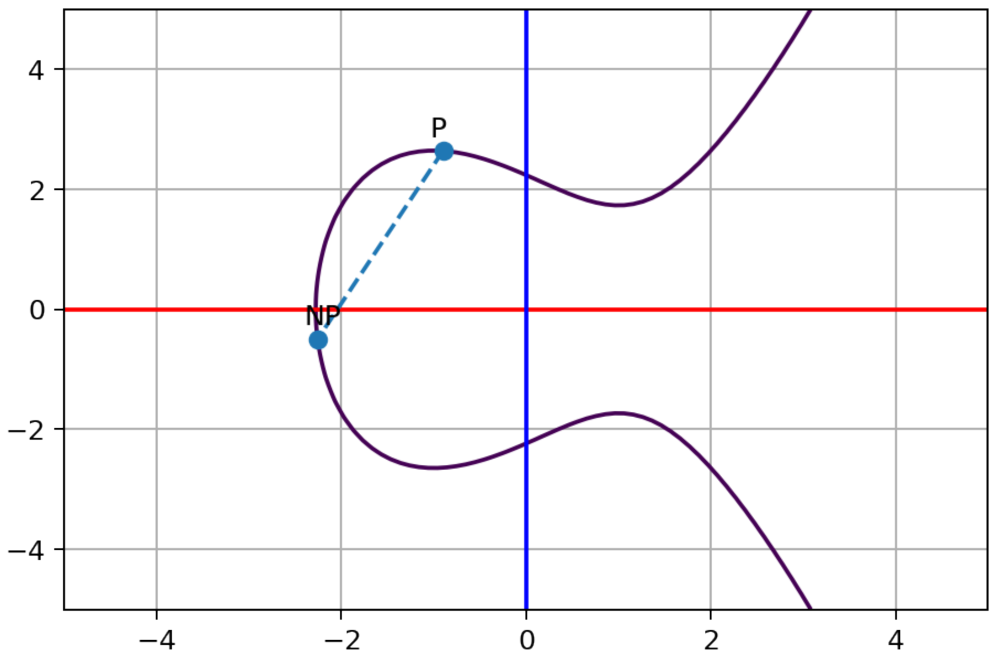
P + 2P on Elliptic Curve



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

The figure on slide 8 shows P + 2P = 3P. Then we can add P again to get 4P and so on.

NP on Elliptic Curve



Credit: Generated based on MIT Licensed code by Fang-Pen Lin

The figure on slide 9 shows NP. In this example N=13. That is, we start with point P, and add P twelve times, resulting in the point 13P.

How is Point Addition used in Elliptic Curve 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.

Shortcut for Calculating NP

- 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: $255 + 255 = 510$

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.

Elliptic Curve with Modular Arithmetic

- 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.

Contents

Applications of ECC

- 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 ECDH, and digital signatures with ECDSA. We will look at ECDH in the following.

Elliptic Curve Diffie-Hellman Key Exchange (algorithm)

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.

Choosing Parameters for ECC

- 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.

Contents

Hash Functions and MACs

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 23 Dec 2021,
hash.tex, r1951

Contents

Hash Functions and MACs

- Hash functions
  - Takes message as input and returns short, unique and random-looking output
  - Different inputs will produce different outputs
  - Also called: MDC, unkeyed hash function
  - Output called: hash (h), digital fingerprint, imprint, message digest
  - $h = H(M)$
- MAC1
  - 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
  - $t = MAC(K, M)$

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*.

Commonly Required Security Properties

- 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.

Security Properties for Selected Applications

- 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

Contents

Hash Functions for Cryptography

- Hash function or algorithm $\mathrm{H}()$:
  - 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.

Applications of Hash Functions

- 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.

Design Approaches for Hash Functions

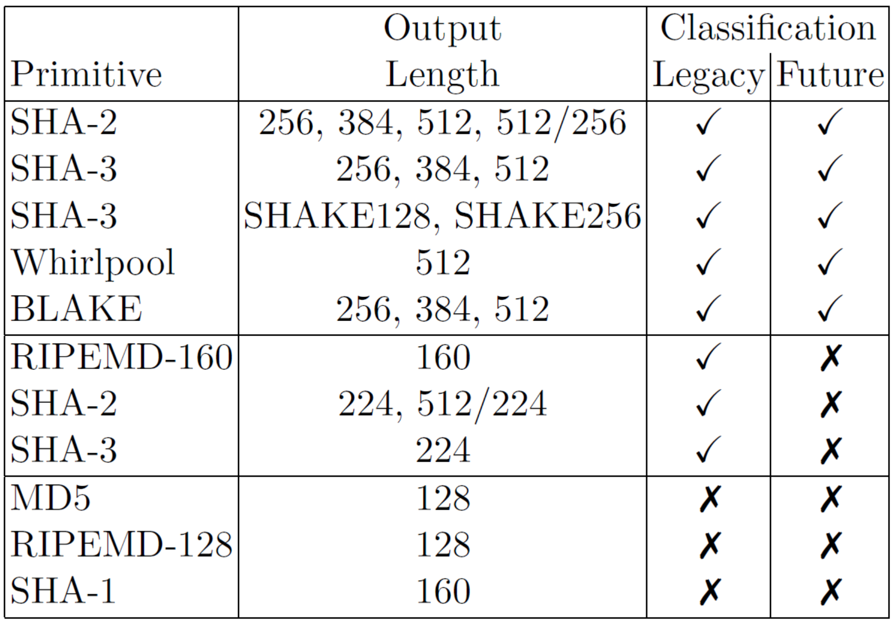
Based on Block Ciphers Well-known and studied block ciphers are used with a mode of operation to produce a hash function. Generally, less efficient than customised hash functions.

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.

Selected Cryptographic Hash Functions



Primitive	Output Length	Classification 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

The figure on slide 10 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.

Contents

Pre-image of a Hash Value (definition)

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.

Hash Collision (definition)

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.

Number of Collisions (exercise)

If H_1 takes fixed length 200-bit messages as input, and produces a 80-bit hash value as output, are collisions possible?

Requirements of Cryptographic Hash Functions

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

Pre-image Resistant Property (definition)

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.

Second Pre-image Resistant Property (definition)

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.

Collision Resistant Property (definition)

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.

Brute Force Attacks on Properties

- 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

Brute Force Attack on Hash Function (exercise)

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.

Contents

Unkeyed and Keyed Hash Functions

- 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

Design Approaches for MACs

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.

Computation Resistance of MAC (definition)

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, $\mathrm{MAC}(K, y)$.

Security of MACs

- 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^kBut ... many tags match T_1For 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 KSimilar effort required as one-way/weak collision resistant property for hash functionsFor n bit MAC value length, effort is 2^n

- Effort to break MAC: $\min(2^k, 2^n)$

Authentication and Data Integrity

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 23 Dec 2021,
auth.tex, r1951

Contents

Attacks on Information Transfer

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.

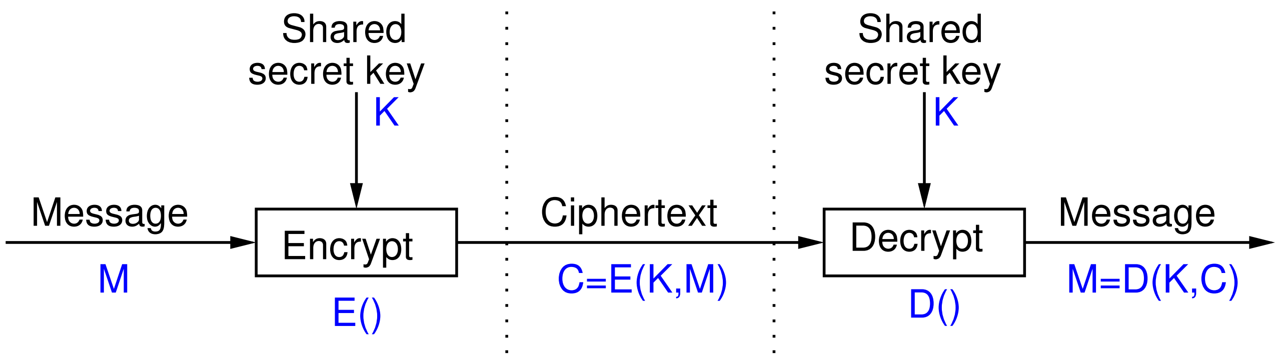
Aims of Authentication

- Receiver wants to verify:
  1. Contents of the message have not been modified (*data authentication*)
  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.

Contents

Symmetric Encryption for Authentication



The figure on slide 6 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.

Recognising Correct Plaintext in English (question)

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).

Recognising Correct Plaintext in English (question)

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.

Recognising Correct Plaintext in Binary (question)

B receives ciphertext (supposedly from A, using shared secret key K):

01101001101011010101101110000010

B decrypts with key K to obtain plaintext:

01011101000011010010101001011110

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.

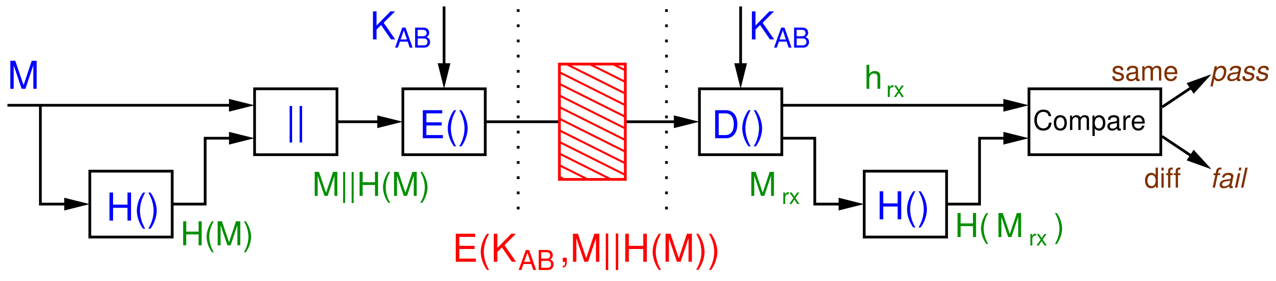
Recognising Correct Plaintext

- 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.

Contents

Authentication by Hash and then Encrypt

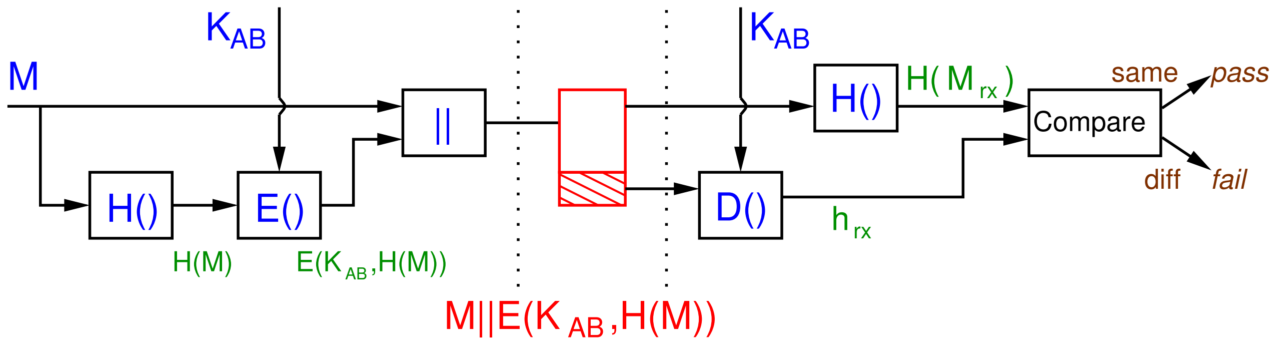


The figure on slide 12 shows a scheme where the hash function is used to add structure to the message. Again, user A and B are on the left and right, respectively. The inputs (message and secret key) and operations are shown in blue. The green values are used to refer to intermediate values. In the middle 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.

Authentication by Encrypting a Hash

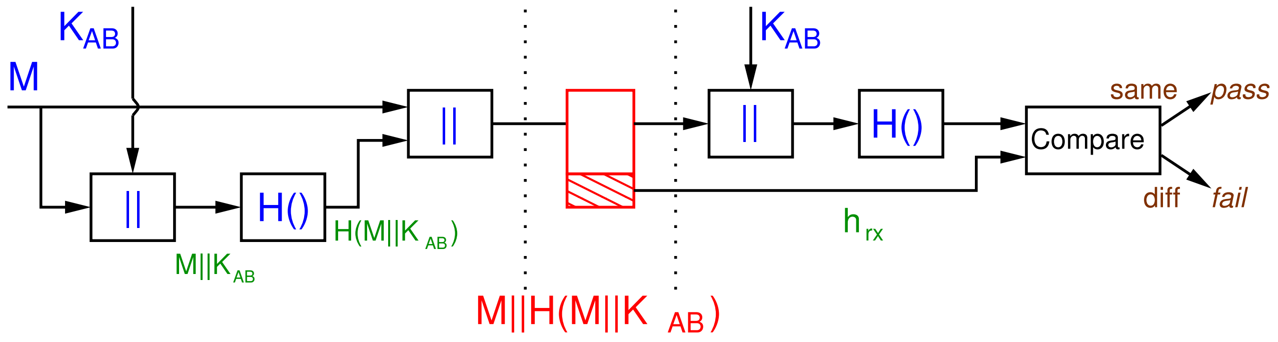


The figure on slide 13 shows a different scheme where only the hash value is encrypted. The receiver can verify that nothing has been changed. This scheme provides authentication, but does not attempt to provide confidentiality. This is useful in reducing any computation overhead when confidentiality is not required.

Attack of Authentication by Encrypting a Hash (exercise)

If a hash function did not have the Second Preimage Resistant property, then demonstrate an attack on the scheme in The figure on slide 13.

Authentication with Hash of a Shared Secret



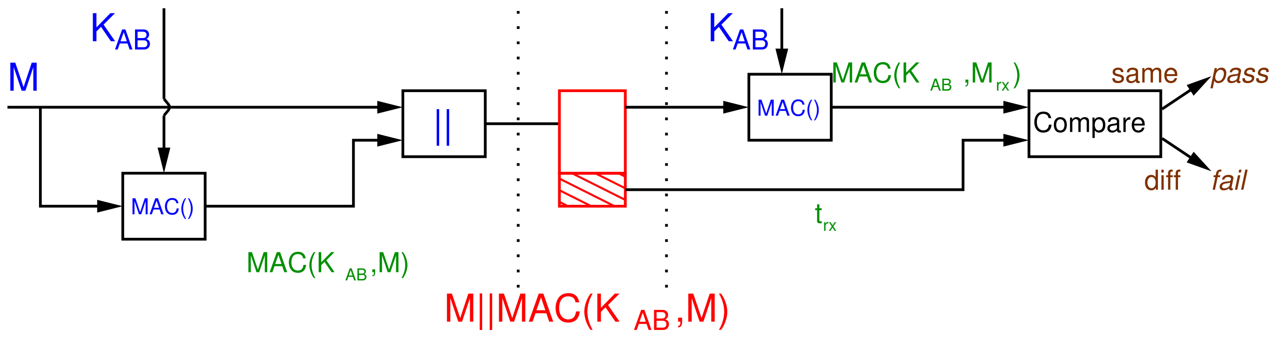
The figure on slide 15 shows a scheme the provides authentication, but without using any encryption. Avoiding encryption can be desirable in very resource constrained environments. *S* is a secret value shared by *A* and *B*. Concatenating the secret with the message, and then hashing the result, allows the receiver the verify the plaintext is correct, and keeps the secret confidential.

Attack of Authentication with Hash of Shared Secret (exercise)

If a hash function did not have the Preimage Resistant property, then demonstrate an attack on the scheme in The figure on slide 15.

Contents

Authentication with only MACs



The figure on slide 18 shows a scheme where authentication is provided using only a MAC. That is, encryption is not used.

Authentication using Encryption and a MAC

- 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
- 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

Encrypt-then-MAC (definition)

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.

MAC-then-Encrypt (definition)

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:

$$\mathrm{E}(K_1, M || \mathrm{MAC}(K_2, M))$$

Encrypt-and-MAC (definition)

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:

$$\mathrm{E}(K_1, M)||\mathrm{MAC}(K_2, M)$$

Recommended Approach for Authenticated Encryption

- 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.

Contents

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).

Digital Signatures in Practice

- 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

Digital Signature Example

User A

Knows (PU_A,PR_A)

1. Sign message M:
   $S = E_{pub}(PR_A, H(M))$
2. Append signature to message and send

$$\xrightarrow{M \| S}$$

User B

Knows PU_A

3. Decrypt
   $h = D_{pub}(PU_A, S)$
4. Compare h with hash of received message

   if H(M) == h
      then message verified
   else
      verification failed
      (don't trust message)

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)

Key Distribution and Management

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
keys.tex, r1969

Contents

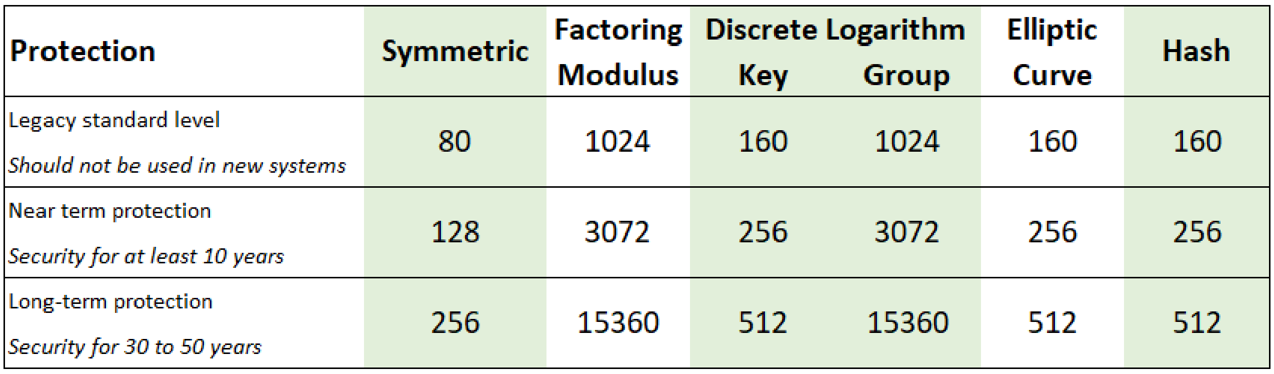
Comparing Key Lengths Across Symmetric and Public Key Algorithms

- 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.

Recommend Key Lengths from ECRYPT-CSA 2018



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

The figure on slide 4 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.

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.

Digital Certificates

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 19 Feb 2020,
cert.tex, r1792

Quantum Computing and Cryptography

Cryptography

School of Engineering and Technology
CQUniversity Australia

Prepared by Steven Gordon on 04 Jan 2022,
quantum.tex, r1971

Contents

Quantum Technology (definition)

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

bit (definition)

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.

qubit (definition)

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.

Quantum Superposition (definition)

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.

qubit Superposition (example)

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.

The Measurement Problem (definition)

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.

Quantum Entanglement (definition)

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.

qubit Entanglement (example)

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.

Quantum Computation (informal) (definition)

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.

Classical Computer Circuits (definition)

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.

Quantum Computer Circuits (definition)

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.

Quantum Computer (definition)

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.

Contents

Quantum Register (definition)

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.

Quantum Parallelism (definition)

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.

Classical Function (example)

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.

Quantum Function (example)

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.

Quantum Algorithm (definition)

A quantum algorithms are usually a combination of classical algorithms/computations and quantum computations. First pre-processing 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.

Grover's Search Algorithm (definition)

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.

Worst Case Brute Force Attempts with Classical and Quantum Algorithms

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}	$\sqrt{2^{512}} = 2^{256}$

The table on slide 22 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.

Integer Factorisation with General Number Field Sieve (definition)

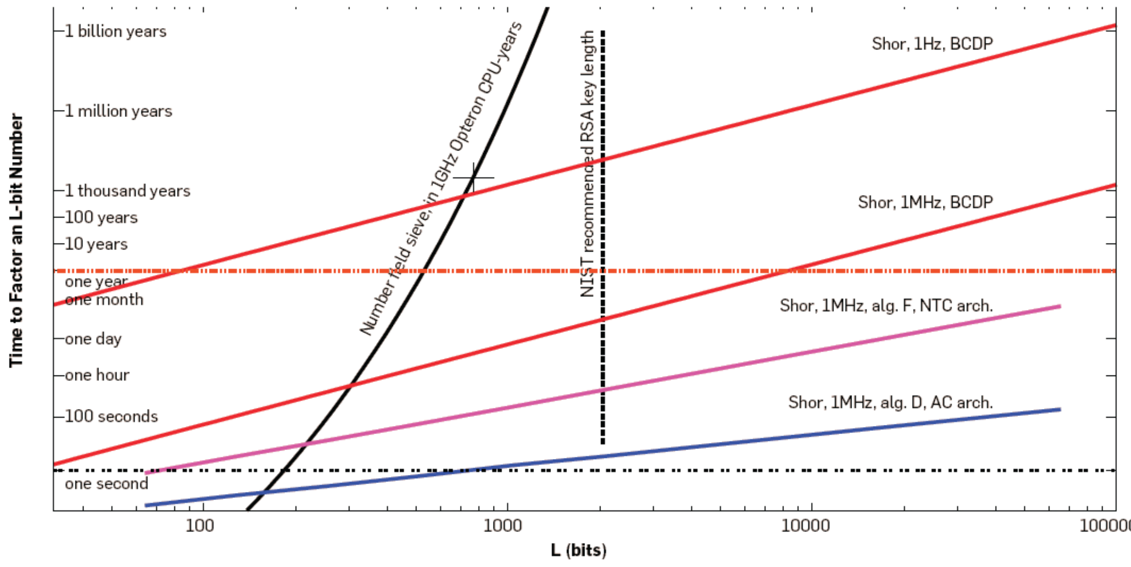
Given an integer N, find its prime factors. A general number field sieve on classical computer takes subexponential time, about $2^{\mathcal{O}(N^{1/3})}$.

Integer Factorisation with Schor's Algorithm (definition)

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.

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

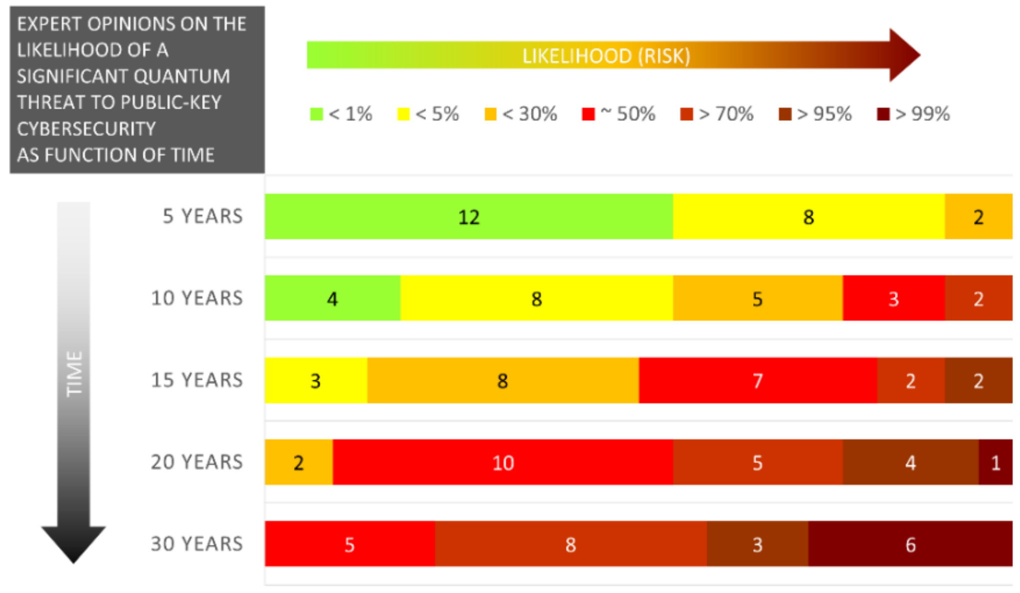


Credit: Figure 1 from A Blueprint For Building a Quantum Computer by Van Meter and Horsman, Communications of the ACM, Oct 2013. Copyright by Van Meter and Horsman and ACM.

The figure on slide 25 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.

Likelihood quantum computers significant threat to public-key cryptosystems



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.

The figure on slide 26, 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.

Contents

Decoherence in Quantum Computing (definition)

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.

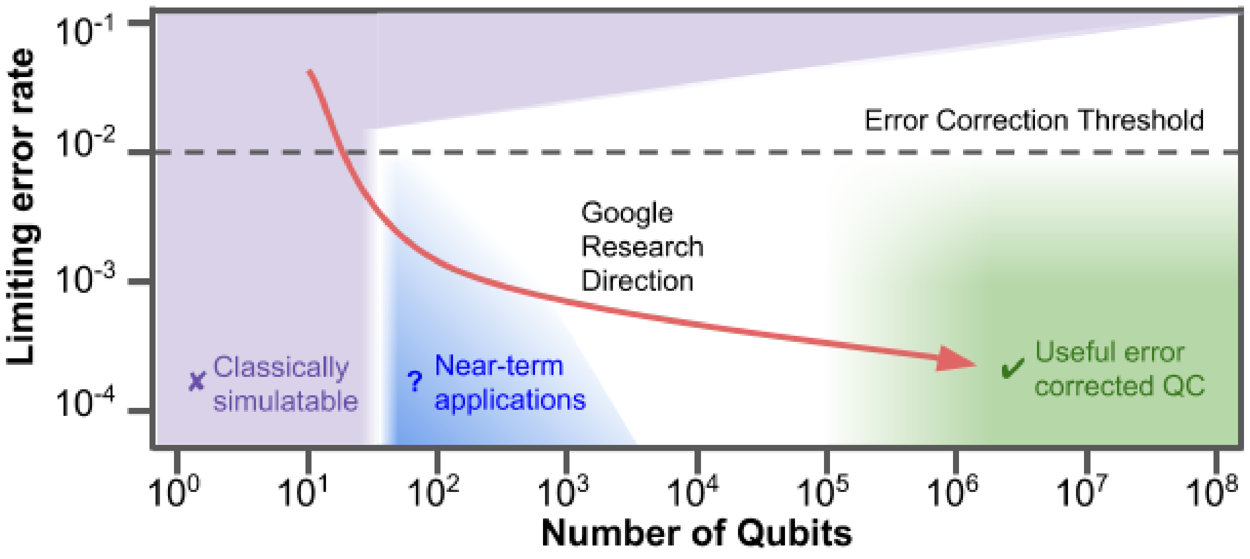
Increasing the time that qubits can hold their coherent state is one practical aim of quantum computing. See the T2 column in the Quantum Computing Report for example values.

Errors in Quantum Computing (definition)

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.

Quantum error rates vs qubits and intended direction of Google Quantum Research



Credit: Google Research, A Preview of Bristlecone, Google's New Quantum Processor

The figure on slide 30, 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.

Cooling (definition)

For qubits to maintain coherence, quantum circuits need to be very cold, approaching 0 Kelvin or -273 C.

Quantum Computers in Practice

- 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

Contents

Quantum Cryptography (definition)

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.

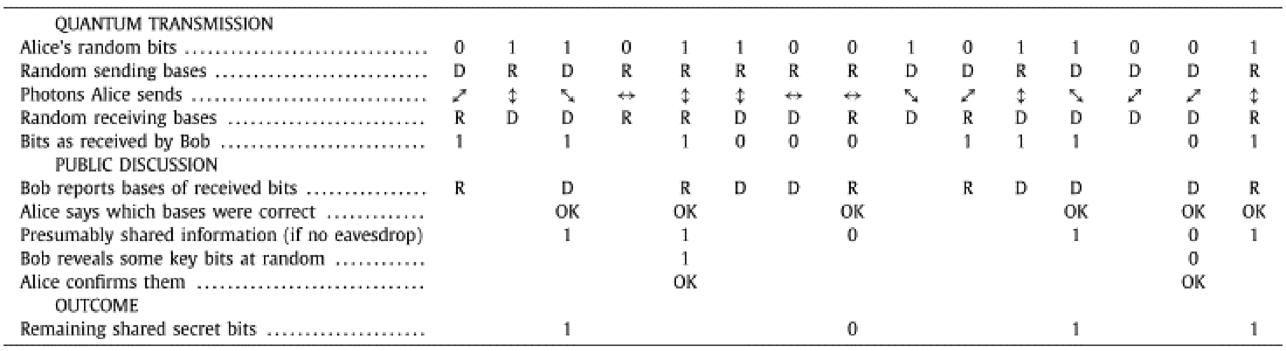
Note that while quantum computers can be used to break cryptographic mechanisms (e.g. using Schor's algorothm), quantum cryptography is separate topic of quantum systems that is about creating cryptographic mechanisms. Quantum cryptographic mechanisms will use quantum computers.

Quantum Key Distribution (informal) (definition)

The aim of 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`.

Example of BB84 Quantum Key Distribution



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.

The figure on slide 36 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.

QKD security (informal) (definition)

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.

Contents

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