CS 458 / 658: Computer Security and Privacy

Module 5 - Security and Privacy of Internet Applications

Part 1 - Basis of cryptography

Meng Xu *(University of Waterloo)*

Winter 2023

Outline

Cryptography

- What is cryptography?

Cryptography

- What is cryptography?
- Related fields:
  - Cryptography ("secret writing"): *Making secret messages*
    - Turning plaintext (an ordinary readable message) into ciphertext (secret messages that are "hard" to read)
  - Cryptanalysis: *Breaking secret messages*
    - Recovering the plaintext from the ciphertext
  - Cryptology is the science that studies these both
- The point of cryptography is to send secure messages over an insecure medium (e.g., the Internet)

The scope of these lectures

- The goal of the cryptography unit in this course is to show you what cryptographic tools exist, and information about using these tools in a secure manner
- We won't be showing you details of how the tools work
  - For that, see CO 487, or chapter 2 of van Oorschot's textbook or chapter 2.3 of Pfleeger's textbook

Cast of characters

When talking about cryptographic schemes, we often use a standard cast of characters

(Honest) communicating parties

Alice Bob Carol Dave

Adversaries

Eve Mallory

Cast of characters

When talking about cryptographic schemes, we often use a standard cast of characters

(Honest) communicating parties

Alice Bob Carol Dave

Adversaries

Eve Mallory

- Eve: A passive eavesdropper, who can listen to any transmitted messages but does not modify them.
- Mallory: An active Man-In-The-Middle, who can listen to, and modify, insert, or delete transmitted messages
- ... (Many more) ..., Trent (trusted third-party), Peggy (prover), Victor (verifier), etc.

Building blocks

Cryptography contains three major types of components

- Confidentiality components
  - Preventing Eve from reading Alice's messages
- Integrity components
  - Preventing Mallory from modifying Alice's messages without being detected
- Authenticity components
  - Preventing Mallory from impersonating Alice

Kerckhoffs' principle

Shannon's maxim: one ought to design systems under the assumption that the enemy will immediately gain full familiarity with them.

- So don't use "secretive" encryption methods
  - Then what do we do?
- Have public algorithms that use a secret key as input
- It's easy to change the key; it's usually just a smallish number

Kerckhoffs's principle: a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge

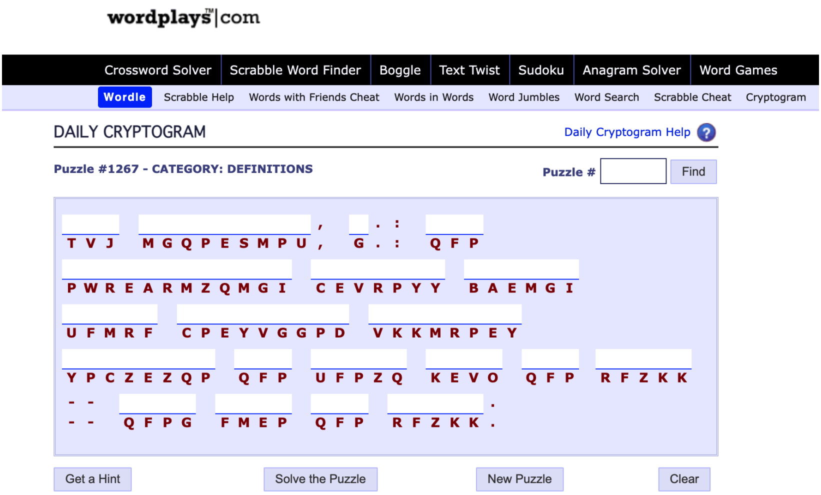
Kerckhoffs' Principle

Kerckhoffs' Principle has a number of implications:

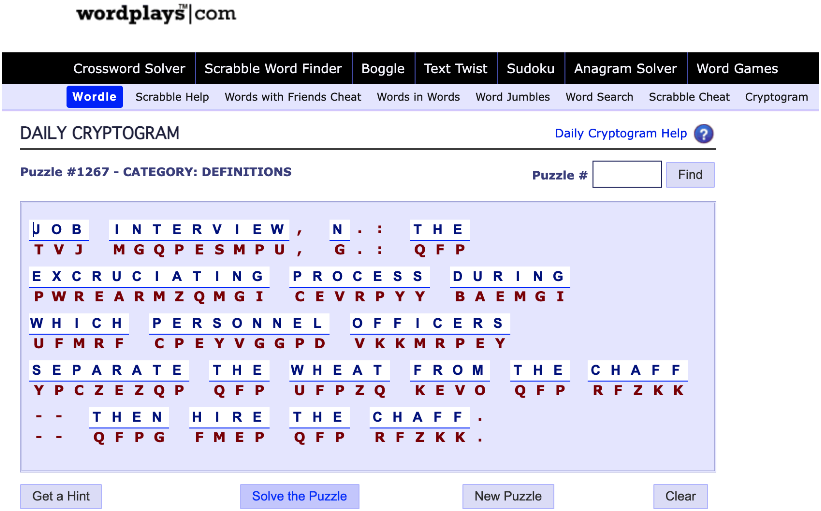
- The system is *at most* as secure as the number of keys
- Eve can just try them all, until she finds the right one
- A strong cryptosystem is one where that's the best Eve can do
  - With weaker systems, there are shortcuts to finding the key

- Example: newspaper cryptogram has 403,291,461,126,605,635,584,000,000 possible keys
- But you don't try them all; it's way easier than that!

Daily cryptogram



Daily cryptogram



Strong cryptosystems

What information do we assume the attacker (Eve) has when she's trying to break our system?

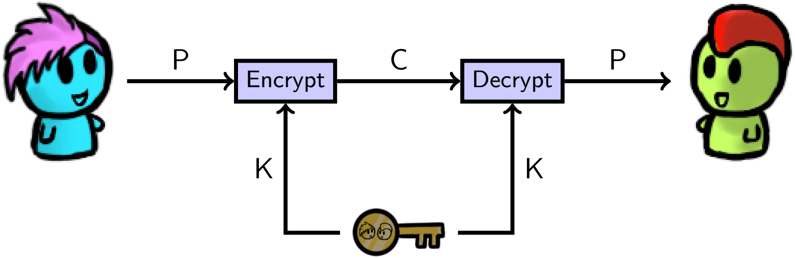
- She may:
  - Know the algorithm
  - Know a number (maybe a large number) of corresponding plaintext/ciphertext pairs
  - Have access to an encryption and/or decryption oracle

And we still want to prevent Eve from learning the key!

Outline

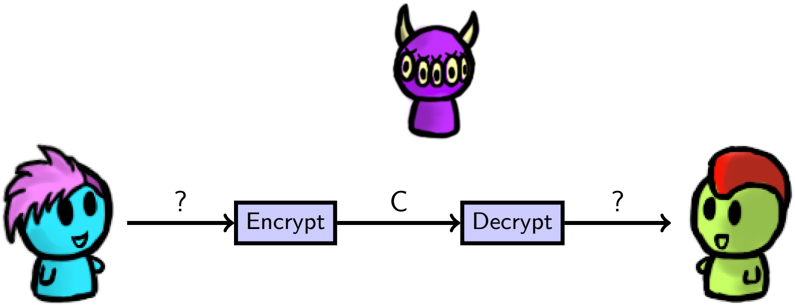
Secret-key encryption

- Secret-key encryption is the simplest form of cryptography
- Used for thousands of years
- Also called symmetric encryption
- The key Alice uses to encrypt the message is the same as the key Bob uses to decrypt it
- $D_k(E_k(m)) = m$



Secret-key encryption

- Eve, not knowing the key, should not be able to recover the plaintext



Vernam cipher

Encrypts one bit at a time by XOR'ing the plaintext with the key:

- Plaintext (t bits): $M = [m_1, m_2, \ldots, m_t]$
- Key (t bits): $K = [k_1, k_2, \ldots, k_t]$
- Ciphertext (t bits):
  $C = [c_1, c_2, \ldots, c_t] = [m_1, m_2, \ldots, m_t] \oplus [k_1, k_2, \ldots, k_t]$

XOR reminder:

$$0 \oplus 0 = 0 \qquad 0 \oplus 1 = 1 \qquad 1 \oplus 0 = 1 \qquad 1 \oplus 1 = 0$$

Q: How do we decrypt?

Vernam cipher

Encrypts one bit at a time by XOR'ing the plaintext with the key:

- Plaintext (t bits): $M = [m_1, m_2, \ldots, m_t]$
- Key (t bits): $K = [k_1, k_2, \ldots, k_t]$
- Ciphertext (t bits):
  $C = [c_1, c_2, \ldots, c_t] = [m_1, m_2, \ldots, m_t] \oplus [k_1, k_2, \ldots, k_t]$

XOR reminder:

$$0 \oplus 0 = 0 \qquad 0 \oplus 1 = 1 \qquad 1 \oplus 0 = 1 \qquad 1 \oplus 1 = 0$$

Q: How do we decrypt?

A: $[m_1, m_2, \ldots, m_t] = [c_1, c_2, \ldots, c_t] \oplus [k_1, k_2, \ldots, k_t]$

One-time pad: definition

If K is randomly chosen and never reused, Vernam cipher is called One-Time Pad

In other words, one-time pad is a secret-key cryptographic scheme with the following construction:

- The key is a truly random bitstring
- The key is of of the same length as the plaintext
- The "Encrypt" and "Decrypt" functions are both XOR

One-time pad: definition

If K is randomly chosen and never reused, Vernam cipher is called One-Time Pad

In other words, one-time pad is a secret-key cryptographic scheme with the following construction:

- The key is a truly random bitstring
- The key is of of the same length as the plaintext
- The "Encrypt" and "Decrypt" functions are both XOR

This provides information-theoretic security.

One-time pad: security

It's very hard to use one-time pad correctly:

- The key must be truly random, not pseudorandom
- The key must be of the same length as the plaintext
- The key (in part or in whole) must never be used more than once
  - A "two-time pad" is insecure!

One-time pad: security

It's very hard to use one-time pad correctly:

- The key must be truly random, not pseudorandom
- The key must be of the same length as the plaintext
- The key (in part or in whole) must never be used more than once
  - A "two-time pad" is insecure!

Q: Why does "try every key" not work here?

One-time pad: security

It's very hard to use one-time pad correctly:

- The key must be truly random, not pseudorandom
- The key must be of the same length as the plaintext
- The key (in part or in whole) must never be used more than once
  - A "two-time pad" is insecure!

Q: Why does "try every key" not work here?

A: Because, given a ciphertext C, for every possible message M, there exists a key K that could have generated that ciphertext.

One-time pad: security

It's very hard to use one-time pad correctly:

- The key must be truly random, not pseudorandom
- The key must be of the same length as the plaintext
- The key (in part or in whole) must never be used more than once
  - A "two-time pad" is insecure!

Q: Why does "try every key" not work here?

A: Because, given a ciphertext C, for every possible message M, there exists a key K that could have generated that ciphertext.

Example:

C = `secret` 01110011 01100101 01100011 01110010 01100101 01110100
K_1 = `------` 00010010 00010001 00010111 00010011 00000110 00011111
M_1 = `attack` 01100001 01110100 01110100 01100001 01100011 01101011
K_2 = `------` 00010111 00000000 00000101 00010111 00001011 00010000
M_2 = `defend` 01100100 01100101 01100110 01100101 01101110 01100100

One-time pad: key sharing

Q: How to share the secret keys?

One-time pad: key sharing

Q: How to share the secret keys?

A: Keys would have to be shared in person or sent by courier or via other secure channels

One-time pad: key sharing

Q: How to share the secret keys?

A: Keys would have to be shared in person or sent by courier or via other secure channels

Q: If the keys are of the same length as the message, what is the point of one-time pad?

One-time pad: key sharing

Q: How to share the secret keys?

A: Keys would have to be shared in person or sent by courier or via other secure channels

Q: If the keys are of the same length as the message, what is the point of one-time pad?

A: The keys can be shared ahead of time

One-time pad: integrity?

Q: Does one-time pad provide integrity?

One-time pad: integrity?

Q: Does one-time pad provide integrity?

Q: If your boss stores your salary (in binary) encrypted with a one time pad, and you have write access to the ciphertext, what can you do with it?

One-time pad: integrity?

Q: Does one-time pad provide integrity?

Q: If your boss stores your salary (in binary) encrypted with a one time pad, and you have write access to the ciphertext, what can you do with it?

A: You can XOR a "10000000000 . . ." (in binary). This flips the most significant bit, which most likely will be zero.

Computational security

In contrast to the "perfect" (or "information-theoretic") security property of one-time pad, most cryptosystems have "computational" security.

- This means that it's certain they can be broken, given *enough* work by Eve
- How much is "enough"?

Computational security

In contrast to the "perfect" (or "information-theoretic") security property of one-time pad, most cryptosystems have "computational" security.

- This means that it's certain they can be broken, given *enough* work by Eve
- How much is "enough"?

At worst, Eve tries every key

- How long that takes depends on how long the keys are
- But it only takes this long if there are no "shortcuts"!

Trying every key: some data points

These are some estimates for RC5:

- One computer can try about 17 million keys per second: $1.7 \cdot 10^7$ keys/second.
- A medium-sized corporate or research lab may have 100 computers: $1.7 \cdot 10^9$ keys/second.
- The Bitcoin network computes 258 million terahashes per second as of Oct 2022. If the hardware could be used to try decrypting with a key in the same time, that's $\approx 2.6 \cdot 10^{20}$ keys/second.

40-bit crypto

This was the US legal export limit for a long time

$2^{40} = 1{,}099{,}511{,}627{,}776$ possible keys

Key size key/second	Computer $\approx 1.7 \cdot 10^7$	Lab $\approx 1.7 \cdot 10^9$	Bitcoin network $\approx 2.6 \cdot 10^{20}$
40-bit	18 hours	11 minutes	4.2 ns

56-bit crypto

This was the US government standard (DES) for a long time

$2^{56} = 72{,}057{,}594{,}037{,}927{,}936$ possible keys

Key size key/second	Computer $\approx 1.7 \cdot 10^7$	Lab $\approx 1.7 \cdot 10^9$	Bitcoin network $\approx 2.6 \cdot 10^{20}$
40-bit	18 hours	11 minutes	4.2 ns
56-bit	134 years	16 months	0.22 ms

128-bit crypto

This is the modern standard

$2^{128} = 340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456$

Key size key/second	Computer $\approx 1.7 \cdot 10^7$	Lab $\approx 1.7 \cdot 10^9$	Bitcoin network $\approx 2.6 \cdot 10^{20}$
40-bit	18 hours	11 minutes	4.2 ns
56-bit	134 years	16 months	0.22 ms
128-bit	$6.3 \cdot 10^{23}$ years	$6.3 \cdot 10^{21}$ years	$4.1 \cdot 10^{10}$ years

128-bit crypto

This is the modern standard

$2^{128} = 340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456$

Key size key/second	Computer $\approx 1.7 \cdot 10^7$	Lab $\approx 1.7 \cdot 10^9$	Bitcoin network $\approx 2.6 \cdot 10^{20}$
40-bit	18 hours	11 minutes	4.2 ns
56-bit	134 years	16 months	0.22 ms
128-bit	$6.3 \cdot 10^{23}$ years	$6.3 \cdot 10^{21}$ years	$4.1 \cdot 10^{10}$ years

To make sense of $4.1 \cdot 10^{10}$ years:

- around 3 times larger than the age of the universe
- around 4.2 times larger than the expected lifetime of the sun.

Well, we cheated a bit

This isn't really true, since computers get faster over time
Moore's law: computing speed doubles every 18 months

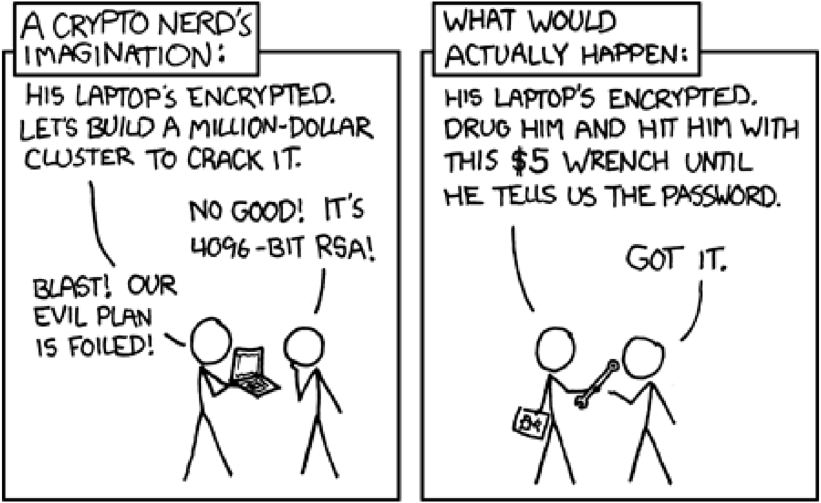
- A better strategy for breaking 128-bit crypto is just to wait until computers get 2^{88} times faster, then break it on one computer in just 18 hours.
- How long do we need to wait? 132 years.
- If we believe Moore's law will keep on working, we'll be able to break 128-bit crypto in 132 years (and 18 hours) :-)
  - Q: Do we believe this?

Well, we cheated a bit

This isn't really true, since computers get faster over time
Moore's law: computing speed doubles every 18 months

- A better strategy for breaking 128-bit crypto is just to wait until computers get 2^{88} times faster, then break it on one computer in just 18 hours.
- How long do we need to wait? 132 years.
- If we believe Moore's law will keep on working, we'll be able to break 128-bit crypto in 132 years (and 18 hours) :-)
  - Q: Do we believe this?
- How about quantum computers? e.g., Grover's algorithm
  - reduces the search space from 2^{128} to 2^{64}
  - requires around 3,000 logical qubits (we have 127 qubits now)

An even better strategy



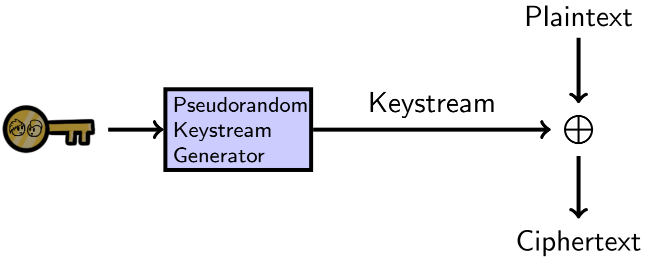
Types of secret-key cryptosystems

Secret-key cryptosystems come in two major classes

- Stream ciphers
- Block ciphers

Stream ciphers

- A stream cipher is what you get if you take the One-Time Pad, but use a pseudorandom keystream instead of a truly random one



- RC4 was the most common stream cipher on the Internet but deprecated. ChaCha is increasingly popular (Chrome and Android), and SNOW3G is mostly used in mobile phone networks.

Two-time pad

Q: What happens if you use the same key (therefore, same keystream) to encrypt two messages?
$C_1 = M_1 \oplus K$, $\quad C_2 = M_2 \oplus K$

Two-time pad

Q: What happens if you use the same key (therefore, same keystream) to encrypt two messages?
$C_1 = M_1 \oplus K\ , \quad C_2 = M_2 \oplus K$

A: We can XOR the ciphertexts:
$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_1 \oplus K) = M_1 \oplus M_2$

Two-time pad

Q: What happens if you use the same key (therefore, same keystream) to encrypt two messages?
$C_1 = M_1 \oplus K \,, \quad C_2 = M_2 \oplus K$

A: We can XOR the ciphertexts:
$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_1 \oplus K) = M_1 \oplus M_2$

Q: Why is this an issue?

Two-time pad

Q: What happens if you use the same key (therefore, same keystream) to encrypt two messages?
$C_1 = M_1 \oplus K\,, \quad C_2 = M_2 \oplus K$

A: We can XOR the ciphertexts:
$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_1 \oplus K) = M_1 \oplus M_2$

Q: Why is this an issue?

A: Messages are not purely random!

Two-time pad, illustrated

C_1

C_2

Two-time pad, illustrated

C_1 C_2 $C_1 \oplus C_2$

Two-time pad, illustrated

C_1

C_2

$C_1 \oplus C_2$

M_2

M_1

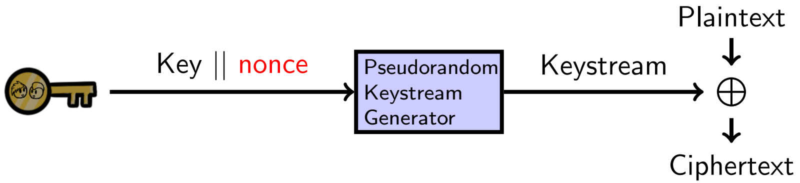
Correct use of stream ciphers

Q: How would you solve this problem without requiring a new shared secret key for each message?

Correct use of stream ciphers

Q: How would you solve this problem without requiring a new shared secret key for each message?

A: Concatenate key with a nonce that is randomly generated for each message and can be send in plaintext



Stream ciphers

- Stream ciphers can be very fast
  - This is useful if you need to send a lot of data securely
- But they can be tricky to use correctly!
  - We saw the issues of re-using a key! (two-time pad)
  - Always remember to pick and random (and never re-use) a nonce

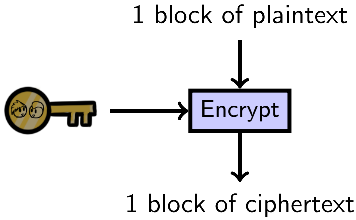
WEP, PPTP are great examples of how not to use stream ciphers.

Block ciphers

- Stream ciphers operate on the message one bit at a time
- An alternative design is block ciphers
  - Block ciphers operate on the message one block at a time
  - Blocks are usually 64 or 128 bits long
- AES is the block cipher everyone should use today
  - Unless you have a really, really good reason
  - Native AES support on Intel chips since Westmere (2010)

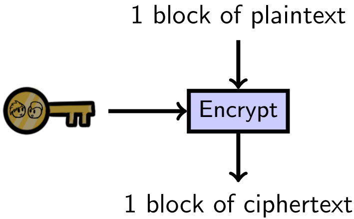
Modes of operation

- Block ciphers work like this:



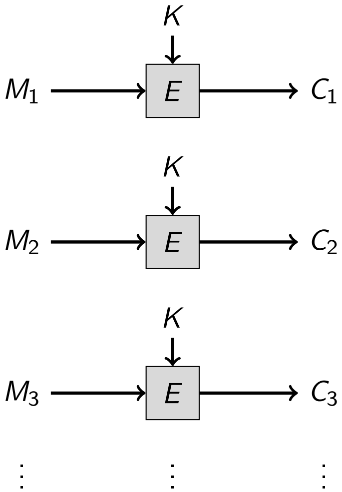
Modes of operation

- Block ciphers work like this:



- If the plaintext is smaller than one block: padding.
- If the plaintext is larger than one block: the choice of what to do with multiple blocks is called the mode of operation of the block cipher.

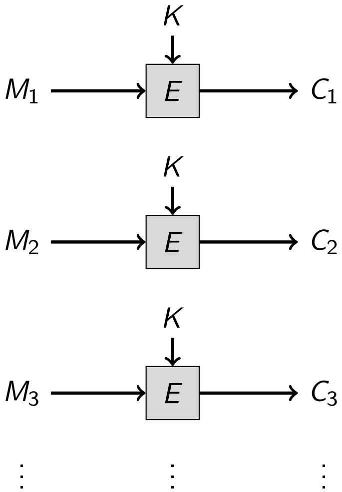
ECB mode



The simplest thing to do is just to encrypt each successive block separately — This is called Electronic Code Book (ECB) mode.

Q: What happens if the plaintext M has some blocks that are identical, $M_i = M_j$?

ECB mode

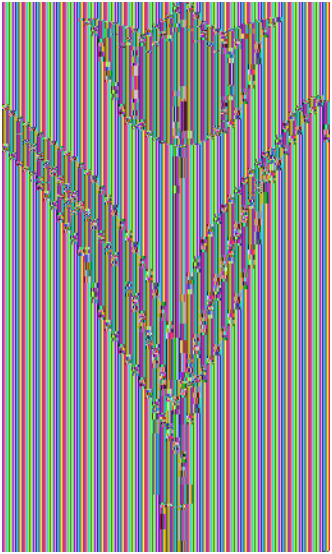


The simplest thing to do is just to encrypt each successive block separately — This is called Electronic Code Book (ECB) mode.

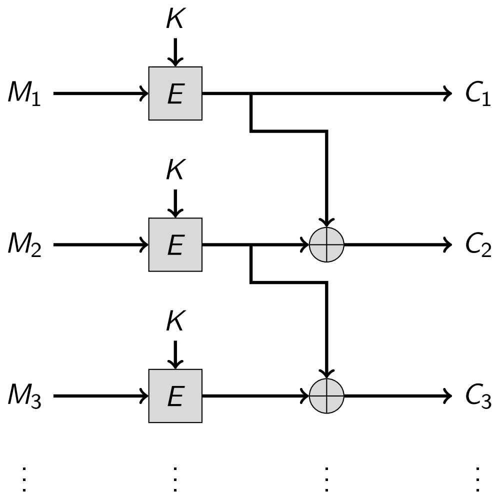
Q: What happens if the plaintext M has some blocks that are identical, $M_i = M_j$?

A: $C_i = E_K(M_i)$, $C_j = E_K(M_j) \implies C_i = C_j$: This reveals patterns in the ciphertext...

ECB mode: example



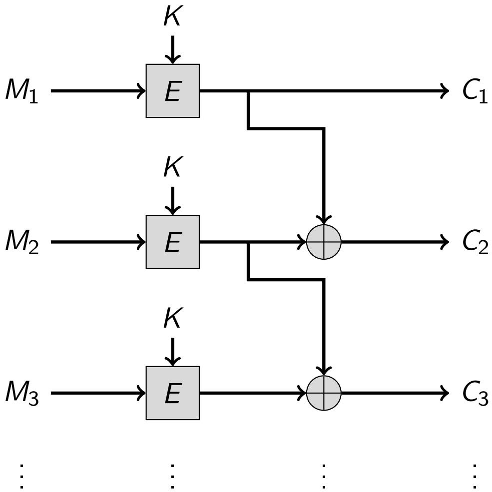
Improving ECB (v1)



We can provide "feedback" among different blocks, to avoid repeating patters.

Q: Does this "feedback" avoid repeating patterns? Any issues here?

Improving ECB (v1)

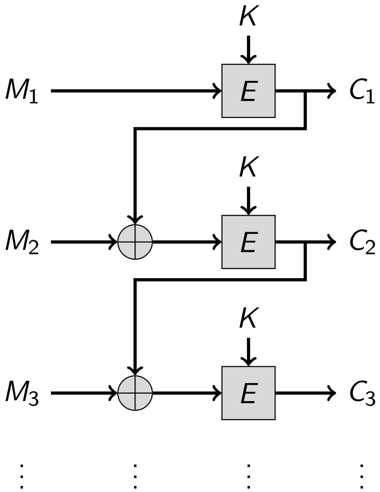


We can provide "feedback" among different blocks, to avoid repeating patters.

Q: Does this "feedback" avoid repeating patterns? Any issues here?

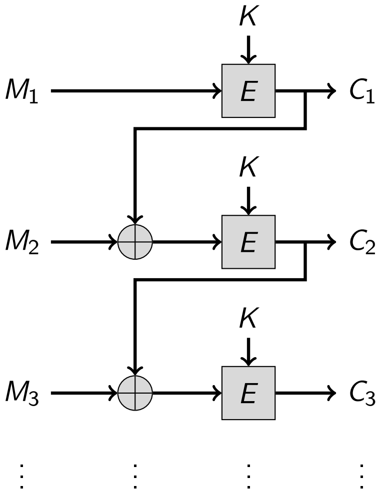
A: We can un-do the XOR if we get all the ciphertexts. This basically does not improve compared to ECB.

Improving ECB (v2)



Q: Does this avoid repeating patterns among blocks? Any issues here?

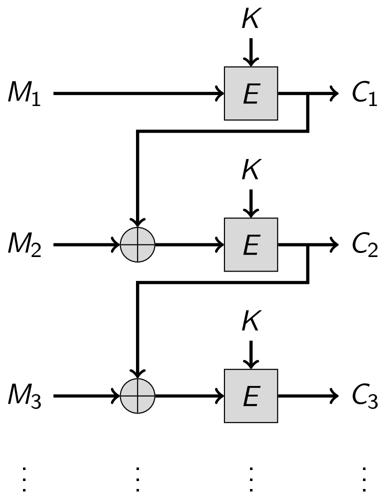
Improving ECB (v2)



Q: Does this avoid repeating patterns among blocks? Any issues here?

Q: What would happen if we encrypt the message twice with the same key?

Improving ECB (v2)



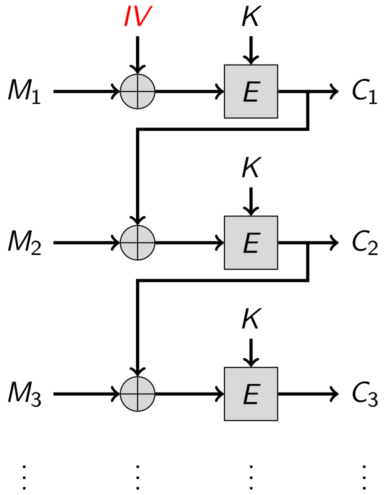
Q: Does this avoid repeating patterns among blocks? Any issues here?

Q: What would happen if we encrypt the message twice with the same key?

A: We get the same ciphertext

To avoid this, we could change the key... but there's a better way

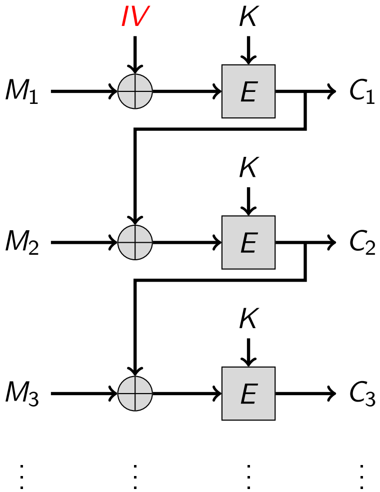
CBC mode



Q: Does this solve the issue of re-encrypting equal blocks?

Q: Does this solve the issue of re-encrypting equal plaintext?

CBC mode



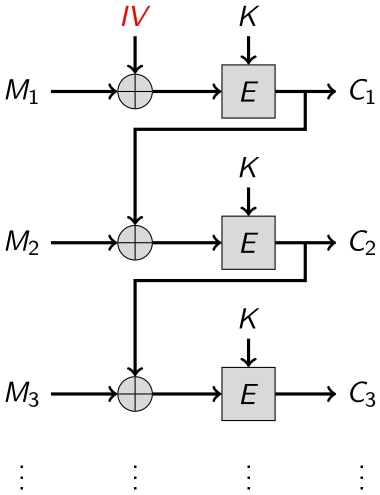
Q: Does this solve the issue of re-encrypting equal blocks?

Q: Does this solve the issue of re-encrypting equal plaintext?

A: Yes! This is called the Cipher-Block Chaining mode

Q: Can we share IV in the clear?

CBC mode



Q: Does this solve the issue of re-encrypting equal blocks?

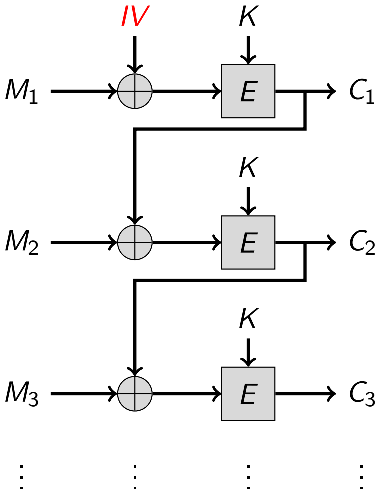
Q: Does this solve the issue of re-encrypting equal plaintext?

A: Yes! This is called the Cipher-Block Chaining mode

Q: Can we share IV in the clear?

A: Yes!!

CBC mode



Q: Does this solve the issue of re-encrypting equal blocks?

Q: Does this solve the issue of re-encrypting equal plaintext?

A: Yes! This is called the Cipher-Block Chaining mode

Q: Can we share IV in the clear?

A: Yes!!

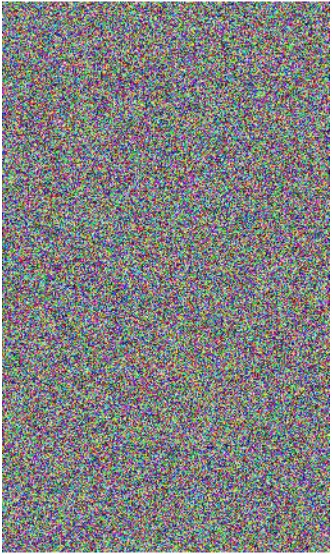
An initialization vector might also be called as a nonce (number used once) or a salt.

Safe modes of operation

There are different modes of operation for block ciphers. Common ones include Cipher Block Chaining (CBC), Counter (CTR), and Galois Counter (GCM) modes

- Patterns in the plaintext are no longer exposed because these modes involves some kind of "feedback" among different blocks
- But you need an IV

CBC mode: example



Key exchange

How do Alice and Bob share the secret key?

- Meet in person
- Diplomatic courier
- ...
- In general this is very hard

Or, we invent new technology...

Outline

Public-key cryptography

- Invented (in public) in the 1970's
- Also called asymmetric cryptography
  - Allows Alice to send a secret message to Bob without any prearranged shared secret!
  - In secret-key cryptography, the same key encrypts the message and also decrypts it
  - In public-key cryptography, there's one key for encryption, and a different key for decryption!
- Some common examples:
  - RSA, ElGamal, ECC, NTRU, McEliece

Public-key cryptography

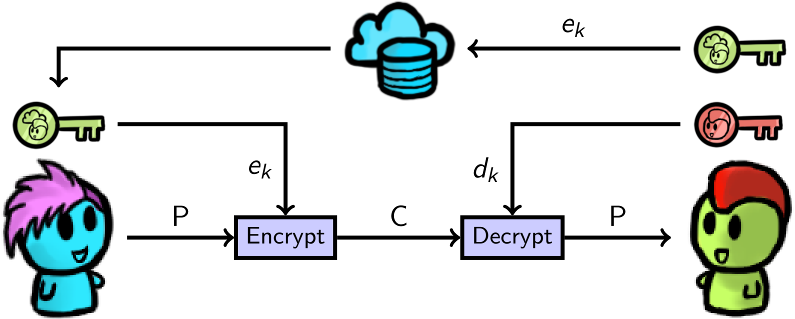
How does it work?

1. Bob creates a key pair (e_k, d_k)
2. Bob gives everyone a copy of his public encryption key e_k
3. Alice uses it to encrypt a message, and sends the encrypted message to Bob
4. Bob uses his private decryption key d_k to decrypt the message
   - Eve can't decrypt it; she only has the encryption key e_k
   - Neither can Alice!
   - It must be hard to derive d_k from e_k

So with this, Alice just needs to know Bob's public key in order to send him secret messages

- These public keys can be published in a directory somewhere

Public-key cryptography



Textbook RSA

- First popular public-key encryption method (published in 1977)
- Relies on the practical difficulty of the factoring problem: given the product of two large prime numbers $n = p \cdot q$, it is computationally hard to factor n.
- Modular arithmetic: integer numbers that “wrap around”
- High-level idea:
  - It is easy to find large integers e, d, and n, such that:

$$(m^e)^d \equiv m \pmod{n}$$

  - But knowing e and n (and even m), it is extremely hard to find d.

Textbook RSA (simplified)

Textbook RSA (simplified)

- Choose two large primes p and q (these are secret).

Textbook RSA (simplified)

- Choose two large primes p and q (these are secret).
- Compute $n = p \cdot q$.
- "Choose" a number e such that $gcd(e, \phi(n)) = 1$ where $\phi(n) = (p-1) \cdot (q-1)$.
- Find d such that $e \cdot d \equiv 1 \pmod{n}$ (This is easy via the *extended Euclidean algorithm*).

Textbook RSA (simplified)

- Choose two large primes p and q (these are secret).
- Compute $n = p \cdot q$.
- "Choose" a number e such that $gcd(e, \phi(n)) = 1$ where $\phi(n) = (p-1) \cdot (q-1)$.
- Find d such that $e \cdot d \equiv 1 \pmod{n}$ (This is easy via the *extended Euclidean algorithm*).
- Public key: (e, n)
- Private key: (d, n)
- Other numbers can be discarded

Textbook RSA (simplified)

- Choose two large primes p and q (these are secret).
- Compute $n = p \cdot q$.
- "Choose" a number e such that $gcd(e, \phi(n)) = 1$ where $\phi(n) = (p-1) \cdot (q-1)$.
- Find d such that $e \cdot d \equiv 1 \pmod{n}$ (This is easy via the *extended Euclidean algorithm*).
- Public key: (e, n)
- Private key: (d, n)
- Other numbers can be discarded
- Encryption: $c \equiv m^e \pmod{n}$
- Decryption: $c^d \pmod{n}$

Textbook RSA (simplified)

- Choose two large primes p and q (these are secret).
- Compute $n = p \cdot q$.
- "Choose" a number e such that $gcd(e, \phi(n)) = 1$ where $\phi(n) = (p-1) \cdot (q-1)$.
- Find d such that $e \cdot d \equiv 1 \pmod{n}$ (This is easy via the *extended Euclidean algorithm*).
- Public key: (e, n)
- Private key: (d, n)
- Other numbers can be discarded
- Encryption: $c \equiv m^e \pmod{n}$
- Decryption: $c^d \pmod{n}$

This is textbook RSA, never do this!! It is not secure.

Example of Textbook RSA

Example (very small RSA):

$p = 53,\ q = 101,\ e = 139,\ d = 1459$

- Compute $n = 53 \cdot 101 = 5353$
- Compute $C_1 = E_e(1011) = 1011^{139} \mod 5353 = 5253$
  - $D_d(5253) = 5253^{1459} \mod 5353 = 1011$
- Compute $C_2 = E_e(4) = 4^{139} \mod 5353 = 324$
  - $D_d(324) = 324^{1459} \mod 5353 = 4$

Feel free to use an online Modular Exponentiation Calculator

Example of Textbook RSA

Q: Compute $D_d(C_1 \cdot C_2)$. What is happening? Why?

Example of Textbook RSA

Q: Compute $D_d(C_1 \cdot C_2)$. What is happening? Why?

A:

$$\begin{aligned} D_d(5253 \cdot 324) =& D_d(1701972) \\ &= 1701972^{1459} \mod 5353 \\ &= 4044 \\ &= 1011 \cdot 4 \end{aligned}$$

The decryption is the product of the original plaintexts.
$(m_1)^e \cdot (m_2)^e \equiv (m_1 \cdot m_2)^e$.

Malleability: it is possible to transform a ciphertext into another ciphertext that decrypts to a related plaintext.
This is typically (but not always!) undesirable.

Chosen ciphertext attack on Textbook RSA

Settings:

- You know Alice's public key (e, n)
- You know some ciphertext c is encrypted with Alice's public key but you don't know the plaintext m
- Alice is willing to decrypt anything for you except for c

Q: What can you do to recover m?

Chosen ciphertext attack on Textbook RSA

Settings:

- You know Alice's public key (e, n)
- You know some ciphertext c is encrypted with Alice's public key but you don't know the plaintext m
- Alice is willing to decrypt anything for you except for c

Q: What can you do to recover m?

A: You can ask Alice to decrypt $(2^e \mod n) \cdot c$

The decryption yields $2 \cdot m$, from which you can recover m

Public key sizes

- Recall that if there are no shortcuts, Eve would have to try 2^{128} things in order to read a message encrypted with a 128-bit symmetric key.
- Unfortunately, all of the public-key methods we know do have shortcuts. For example:
  - Eve could read a message encrypted with a 128-bit RSA key with just 2^{33} work, which is easy!
    - In RSA, $n = pq$; n is public; factoring n reveals the key
    - 2^{33} is the "work factor" to factor a 128-bit integer n
    - Quantum computers can factor even faster, see Shor's algorithm
  - If we want Eve to have to do 2^{128} work, we need to use a much longer public key

Public key sizes

Comparison of key sizes for roughly equal strength

AES	RSA	ECC
80	1024	160
116	2048	232
128	2600	256
160	4500	320
256	14000	512

Hybrid cryptography

- **Secret-key cryptography**: shorter keys, faster, same key to encrypt and decrypt, but requires pre-sharing of the keys.
- **Public-key cryptography**: longer keys, slower, different key to encrypt and decrypt, but does not require sharing of secrets.

Hybrid cryptography

- **Secret-key cryptography**: shorter keys, faster, same key to encrypt and decrypt, but requires pre-sharing of the keys.
- **Public-key cryptography**: longer keys, slower, different key to encrypt and decrypt, but does not require sharing of secrets.

We can get the best of both worlds:

- Pick a random 128-bit key K for a secret-key cryptosystem
- Encrypt the large message with the key K (e.g., using AES)
- Encrypt the key K using a public-key cryptosystem
- Send both the encrypted message and the encrypted key to Bob

Hybrid cryptography

- **Secret-key cryptography**: shorter keys, faster, same key to encrypt and decrypt, but requires pre-sharing of the keys.
- **Public-key cryptography**: longer keys, slower, different key to encrypt and decrypt, but does not require sharing of secrets.

We can get the best of both worlds:

- Pick a random 128-bit key K for a secret-key cryptosystem
- Encrypt the large message with the key K (e.g., using AES)
- Encrypt the key K using a public-key cryptosystem
- Send both the encrypted message and the encrypted key to Bob

This hybrid approach is used for almost every cryptography application on the Internet today

Is that all there is?

It seems we've got this "sending secret messages" thing down pat. What else is there to do?

- Even if we're safe from Eve reading our messages, there's still the matter of Mallory
- It turns out that even if our messages are encrypted, Mallory can sometimes modify them in transit!
- Mallory won't necessarily know what the message says, but can still change it in an undetectable way
  - e.g. bit-flipping attack on stream ciphers
- This is counterintuitive, and often forgotten

How do we make sure that Bob gets the same message Alice sent?

Outline

Integrity components

How do we tell if a message has changed in transit?

Integrity components

How do we tell if a message has changed in transit?

Simplest answer: use a checksum

- For example, add up all the bytes of a message
- The last digits of serial numbers (credit card, ISBN, etc.) are usually checksums

Integrity components

How do we tell if a message has changed in transit?

Simplest answer: use a checksum

- For example, add up all the bytes of a message
- The last digits of serial numbers (credit card, ISBN, etc.) are usually checksums

A naive checksum procedure works like following:

- Alice computes the checksum of the message, and sticks it at the end before encrypting it to Bob.
- When Bob receives the message and checksum, he verifies that the checksum is correct

Simple checksums do not work!

Reason 1: Mallory can simply craft a new message and calculate the checksum of the new message and send both to Bob.

Simple checksums do not work!

Reason 1: Mallory can simply craft a new message and calculate the checksum of the new message and send both to Bob.

Reason 2: Simple checksums are insecure even when the checksum value cannot be changed.

- With most checksum methods, Mallory can easily change the message in such a way that the checksum stays the same
- We need a "cryptographic" checksum
- It should be hard for Mallory to find a second message with the same checksum as any given one

Cryptographic hash functions

A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest

- Common examples: MD5, SHA-1, SHA-2, SHA-3 (a.k.a., Keccak, from 2012 on)

Cryptographic hash functions

A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest

- Common examples: MD5, SHA-1, SHA-2, SHA-3 (a.k.a., Keccak, from 2012 on)

Hash functions should have three properties:

Cryptographic hash functions

A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest

- Common examples: MD5, SHA-1, SHA-2, SHA-3 (a.k.a., Keccak, from 2012 on)

Hash functions should have three properties:

1. Preimage-resistance:
   - Given y, it's hard to find x such that $h(x) = y$
     i.e., a "preimage" of y

Cryptographic hash functions

A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest

- Common examples: MD5, SHA-1, SHA-2, SHA-3 (a.k.a., Keccak, from 2012 on)

Hash functions should have three properties:

1. Preimage-resistance:
   - Given y, it's hard to find x such that $h(x) = y$
     i.e., a "preimage" of y
2. Second preimage-resistance:
   - Given x, it's hard to find $x' \neq x$ such that $h(x) = h(x')$
     i.e., a "second preimage" of $h(x)$

Cryptographic hash functions

A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest

- Common examples: MD5, SHA-1, SHA-2, SHA-3 (a.k.a., Keccak, from 2012 on)

Hash functions should have three properties:

1. Preimage-resistance:
   - Given y, it's hard to find x such that $h(x) = y$
     i.e., a "preimage" of y
2. Second preimage-resistance:
   - Given x, it's hard to find $x' \neq x$ such that $h(x) = h(x')$
     i.e., a "second preimage" of $h(x)$
3. Collision-resistance:
   - It's hard to find any two distinct values x, x' such that $h(x) = h(x')$
     i.e., a "collision"

What is "hard"?

- For SHA-1, for example, it takes 2^{160} work to find a preimage or second preimage, and 2^{80} work to find a collision using a brute-force search
  - However, there are faster ways than brute force to find collisions in SHA-1 or MD5

What is “hard”?

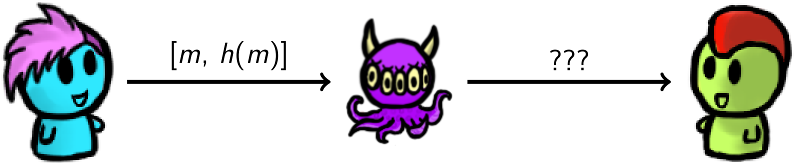
- For SHA-1, for example, it takes 2^{160} work to find a preimage or second preimage, and 2^{80} work to find a collision using a brute-force search
  - However, there are faster ways than brute force to find collisions in SHA-1 or MD5
- Collisions are always easier to find than preimages or second preimages due to the well-known birthday paradox

What is “hard”?

- For SHA-1, for example, it takes 2^{160} work to find a preimage or second preimage, and 2^{80} work to find a collision using a brute-force search
  - However, there are faster ways than brute force to find collisions in SHA-1 or MD5

- Collisions are always easier to find than preimages or second preimages due to the well-known birthday paradox
  - * If there are *n* people in a room, what is the probability that at least two people have the same birthday?
  - For 23 people, the probability is larger than 50%!
  - For 40 people, it's almost 90%!!
  - For 60 people, it's more than 99%!!!

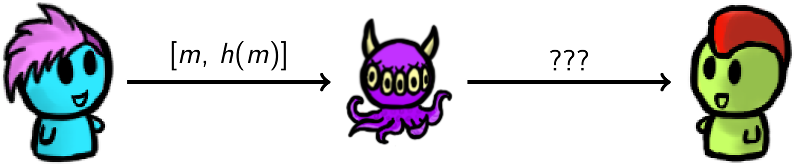
Let's use a hash function!



Assume we don't care about confidentiality, just integrity.

Q: What can Mallory do to change the message?

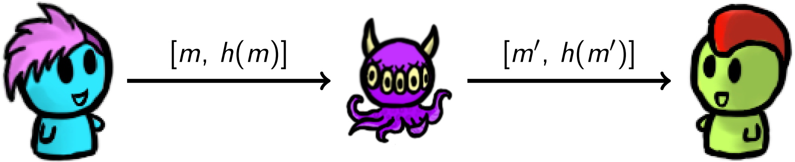
Let's use a hash function!



Assume we don't care about confidentiality, just integrity.

Q: What can Mallory do to change the message?

A: Just change it and compute the new message digest herself!

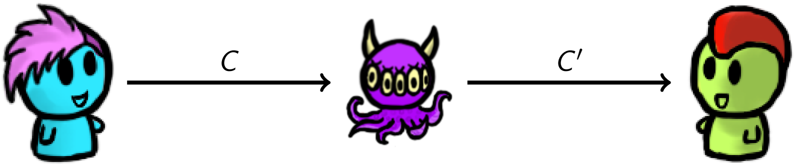


Cryptographic hash functions

- Hash functions provide integrity guarantees only when there is a secure way of sending the message digest
  - For example, Bob can publish a hash of his public key (i.e., a message digest) on his business card
  - Putting the whole key on there would be too big
  - But Alice can download Bob's key from the Internet, hash it herself, and verify that the result matches the message digest on Bob's card
- What if there's no external channel to be had?
  - For example, you're using the Internet to communicate

Outline

How to authenticate the message?



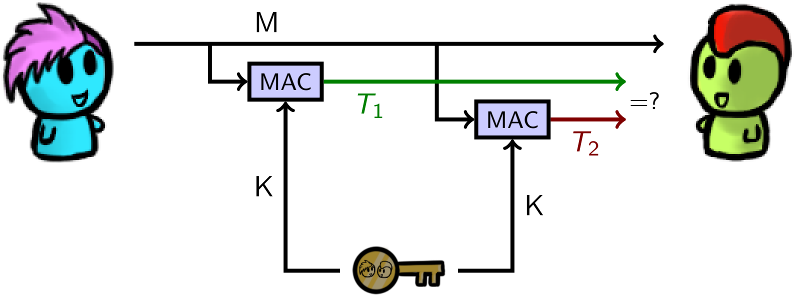
Message authentication codes (MAC)

Assume Alice and Bob share a secret that is only known to them.

We do the following “trick” (a mental model):

- Suppose there exists a large collection of hash functions.
- Alice and Bob can use the secret to pick the “correct” one
- Only those who know the secret can generate, or even check, the computed hash value (sometimes called a tag)
- These “keyed hash functions” are usually called Message Authentication Codes, or MACs
- Common examples:
  - SHA-1-HMAC, SHA-256-HMAC, CBC-MAC

Message authentication codes (MAC)



Combining ciphers and MACs

In practice we often need both confidentiality and message integrity

Combining ciphers and MACs

In practice we often need both confidentiality and message integrity

- There are multiple strategies to combine a cipher and a MAC when processing a message
  - Encrypt-then-MAC, MAC-then-Encrypt, Encrypt-and-MAC

Combining ciphers and MACs

In practice we often need both confidentiality and message integrity

- There are multiple strategies to combine a cipher and a MAC when processing a message
  - Encrypt-then-MAC, MAC-then-Encrypt, Encrypt-and-MAC
- Ideally your crypto library already provides an authenticated encryption mode that securely combines the two operations so you don't have to worry about getting it right
  - E.g., GCM, CCM (used in WPA2, see later), or OCB mode

Combining Ciphers and MACs. Let's try it!

Alice and Bob have a secret key K for a secret-key cryptosystem $(E_K(\cdot), D_K(\cdot))$ and a secret key K' for their MAC $(MAC_{K'}(\cdot))$. Concatenation is $||$. How does Alice build a message for Bob in the following scenarios?

- MAC-then-Encrypt: compute the MAC on the message, then encrypt the message and MAC together, and send that ciphertext.
- Encrypt-and-MAC: compute the MAC on the message, compute the encryption of the message, and send both.
- Encrypt-then-MAC: encrypt the message, compute the MAC on the encryption, send encrypted message and MAC.

Combining Ciphers and MACs. Let's try it!

Alice and Bob have a secret key K for a secret-key cryptosystem $(E_K(\cdot), D_K(\cdot))$ and a secret key K' for their MAC $(MAC_{K'}(\cdot))$. Concatenation is $||$. How does Alice build a message for Bob in the following scenarios?

- MAC-then-Encrypt: compute the MAC on the message, then encrypt the message and MAC together, and send that ciphertext.

$$E_K(m||MAC_{K'}(m))$$

- Encrypt-and-MAC: compute the MAC on the message, compute the encryption of the message, and send both.

$$E_K(m)||MAC_{K'}(m)$$

- Encrypt-then-MAC: encrypt the message, compute the MAC on the encryption, send encrypted message and MAC.

$$E_K(m)||MAC_{K'}(E_K(m))$$

Encrypt and authenticate: what's the right order?

- Usually, we want the receiver to verify the MAC first!

Q: Which of this is the recommended strategy, then?

$E_K(m||MAC_{K'}(m))$ $\quad E_K(m)||MAC_{K'}(m)$ $\quad E_K(m)||MAC_{K'}(E_K(m))$

Encrypt and authenticate: what's the right order?

- Usually, we want the receiver to verify the MAC first!

Q: Which of this is the recommended strategy, then?

$E_K(m||MAC_{K'}(m))$ $\quad E_K(m)||MAC_{K'}(m)$ $\quad E_K(m)||MAC_{K'}(E_K(m))$

A: The recommended strategy is Encrypt-then-MAC:

$$E_K(m)||MAC_{K'}(E_K(m))$$

Encrypt and authenticate: what's the right order?

- Usually, we want the receiver to verify the MAC first!

Q: Which of this is the recommended strategy, then?

$E_K(m||MAC_{K'}(m))$ $\quad$ $E_K(m)||MAC_{K'}(m)$ $\quad$ $E_K(m)||MAC_{K'}(E_K(m))$

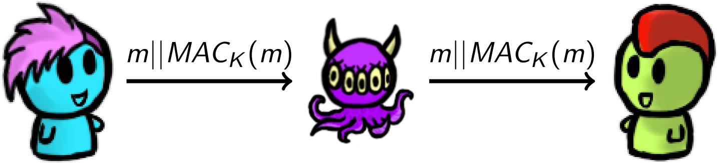
A: The recommended strategy is Encrypt-then-MAC:

$$E_K(m)||MAC_{K'}(E_K(m))$$

- There is a nice blog post that calls this the "Doom principle": if you have to perform *any* cryptographic operation before verifying the MAC on a message you've received, it will *somehow* inevitably lead to doom.
- It explains two simple attacks that can happen if you violate the Doom principle.

Repudiation

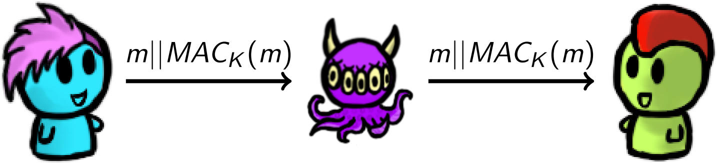
Suppose Alice and Bob share a MAC key K, and Bob receives a message m along with a valid tag $T = MAC_K(m)$.



- Bob can be assured that Alice is the one who sent m and that the message has not been modified since she sent it!
- This is like a “signature” on the message... but not quite!

Repudiation

Suppose Alice and Bob share a MAC key K, and Bob receives a message m along with a valid tag $T = MAC_K(m)$.

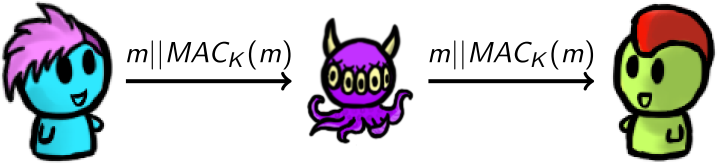


- Bob can be assured that Alice is the one who sent m and that the message has not been modified since she sent it!
- This is like a “signature” on the message... but not quite!
- Bob can't prove to Carol that Alice sent m, though.

Q: Why not?

Repudiation

Suppose Alice and Bob share a MAC key K, and Bob receives a message m along with a valid tag $T = MAC_K(m)$.

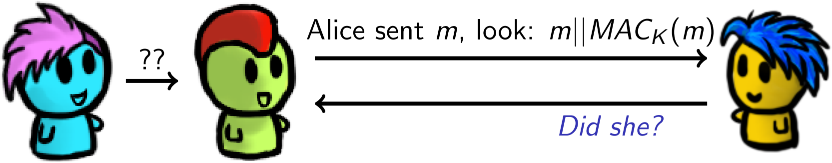


- Bob can be assured that Alice is the one who sent m and that the message has not been modified since she sent it!
- This is like a “signature” on the message... but not quite!
- Bob can't prove to Carol that Alice sent m, though.

Q: Why not?

A: Either Alice or Bob could create any of the message and MAC combinations. Also, Carol doesn't know the secret keys.

Repudiation



- Alice can just claim that Bob made up the message m, and calculated the tag T himself
- This is called repudiation, and we sometimes want to avoid it
- Some interactions should be repudiable
  - Private conversations
- Some interactions should be non-repudiable
  - Electronic commerce

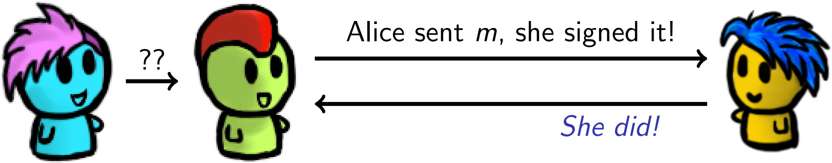
Digital signatures

For non-repudiation, what we want is a true digital signature, with the following properties:

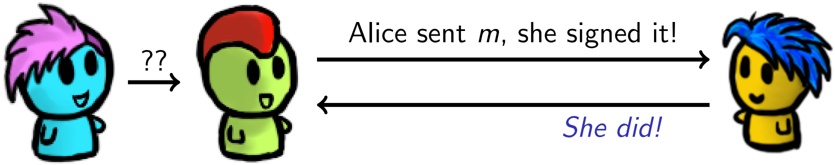
If Bob receives a message with Alice's digital signature on it, then:

- it must be Alice, and not an impersonator, who sent the message (like a MAC)
- the message has not been altered after it was sent (like a MAC),
- Bob can prove these facts to a third party (additional property not satisfied by a MAC).

Digital signatures



Digital signatures



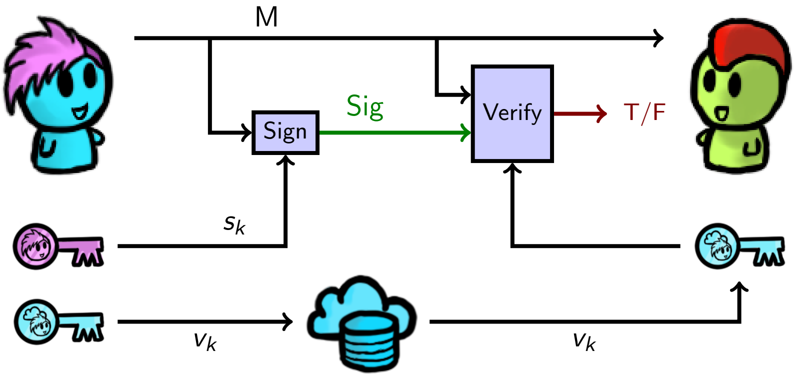
How do we arrange this?

- Use similar techniques to public-key cryptography

Making digital signatures

- Remember public-key cryptosystems:
  - Separate keys for encryption and decryption
  - Give everyone a copy of the encryption key
  - The decryption key is private
- To make a digital signature:
  - Alice signs the message with her private signature key (s_k)
- To verify Alice's signature:
  - Bob verifies the message with Alice's public verification key (v_k)
  - If it verifies correctly, the signature is valid

Making digital signatures



Hybrid signatures

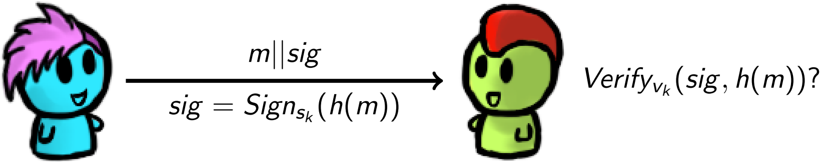
- Just like encryption in public-key cryptosystems, signing large messages is slow
- We can also hybridize signatures to make them faster:
  - Alice sends the (unsigned) message, and also a signature on a hash of the message
  - The hash is much smaller than the message, so it is faster to sign and verify

$$\xrightarrow[sig \,=\, Sign_{s_k}(h(m))]{m || sig}$$

$Verify_{v_k}(sig, h(m))?$

Hybrid signatures

- Just like encryption in public-key cryptosystems, signing large messages is slow
- We can also hybridize signatures to make them faster:
  - Alice sends the (unsigned) message, and also a signature on a hash of the message
  - The hash is much smaller than the message, so it is faster to sign and verify



Remember that authenticity and confidentiality are separate; if you want both, you need to do both

Combining public-key encryption and digital signatures

- Alice has two different key pairs:
  - an (encryption, decryption) key pair (e_k^A, d_k^A)
  - a (signature, verification) key pair (s_k^A, v_k^A)
- So does Bob:
  - an (encryption, decryption) key pair (e_k^B, d_k^B)
  - a (signature, verification) key pair (s_k^B, v_k^B)

Combining public-key encryption and digital signatures

- Alice has two different key pairs:
  - an (encryption, decryption) key pair (e_k^A, d_k^A)
  - a (signature, verification) key pair (s_k^A, v_k^A)
- So does Bob:
  - an (encryption, decryption) key pair (e_k^B, d_k^B)
  - a (signature, verification) key pair (s_k^B, v_k^B)

Q: What would be the best scheme to encode a message m?

- Sign-then-Encrypt: $E_{e_k^B}(m \parallel Sign_{s_k^A}(m))$
- Encrypt-then-Sign: $E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m))$

Combining public-key encryption and digital signatures

- Sign-then-Encrypt: $E_{e_k^B}(m \parallel Sign_{s_k^A}(m))$
- Encrypt-then-Sign: $E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m))$

Q: What can Eve learn from an Encrypt-then-Sign message that she cannot learn from a Sign-then-Encrypt message?

Combining public-key encryption and digital signatures

- Sign-then-Encrypt: $E_{e_k^B}(m \parallel Sign_{s_k^A}(m))$
- Encrypt-then-Sign: $E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m))$

Q: What can Eve learn from an Encrypt-then-Sign message that she cannot learn from a Sign-then-Encrypt message?

A: Eve can see Alice signed the encrypted message (if she has Alice's verification key)

Combining public-key encryption and digital signatures

- Sign-then-Encrypt: $E_{e_k^B}(m \parallel Sign_{s_k^A}(m))$
- Encrypt-then-Sign: $E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m))$

Q: What can Mallory do with a captured Encrypt-then-Sign message?

Combining public-key encryption and digital signatures

- Sign-then-Encrypt: $E_{e_k^B}(m \parallel Sign_{s_k^A}(m))$
- Encrypt-then-Sign: $E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m))$

Q: What can Mallory do with a captured Encrypt-then-Sign message?

A: Mallory could remove the signature and sign it herself! (even if she does not know the plaintext)

$$E_{e_k^B}(m) \parallel Sign_{s_k^A}(E_{e_k^B}(m)) \to E_{e_k^B}(m) \parallel Sign_{s_k^M}(E_{e_k^B}(m))$$

The key management problem

One of the hardest problems of public-key cryptography is that of key management

How can Bob find Alice's verification key?

The key management problem

One of the hardest problems of public-key cryptography is that of key management

How can Bob find Alice's verification key?

- He can know it personally (manual keying)
  - SSH does this

The key management problem

One of the hardest problems of public-key cryptography is that of key management

How can Bob find Alice's verification key?

- He can know it personally (manual keying)
  - SSH does this
- He can trust a friend to tell him (web of trust)
  - PGP does this

The key management problem

One of the hardest problems of public-key cryptography is that of key management

How can Bob find Alice's verification key?

- He can know it personally (manual keying)
  - SSH does this
- He can trust a friend to tell him (web of trust)
  - PGP does this
- He can trust some third party to tell him (CA)
  - TLS / SSL do this

The key management problem

One of the hardest problems of public-key cryptography is that of key management

How can Bob find Alice's verification key?

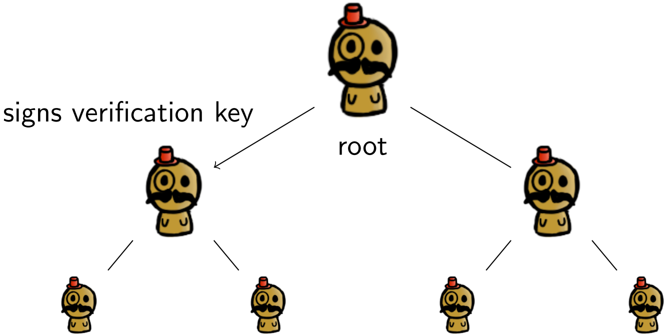
- He can know it personally (manual keying)
  - SSH does this
- He can trust a friend to tell him (web of trust)
  - PGP does this
- He can trust some third party to tell him (CA)
  - TLS / SSL do this
- He trusts no one... (blockchain maybe?)
  - Decentralized Public-Key Infrastructure?

Certificate authorities

- A CA is a trusted third party who keeps a directory of people's (and organizations') verification keys
- Alice generates a (signature, verification) key pair, and sends the verification key, as well as a bunch of personal information, both signed with Alice's signature key, to the CA
- The CA ensures that the personal information and Alice's signature are correct
- The CA generates a certificate consisting of Alice's personal information, as well as her verification key. The entire certificate is signed with the CA's signature key
- https://letsencrypt.org/ has changed the game. Extended validation certificates (for which CAs charged a lot of money) are not treated differently by most browsers after 2019. See more on Extended Validation Certificates are (Really, Really) Dead

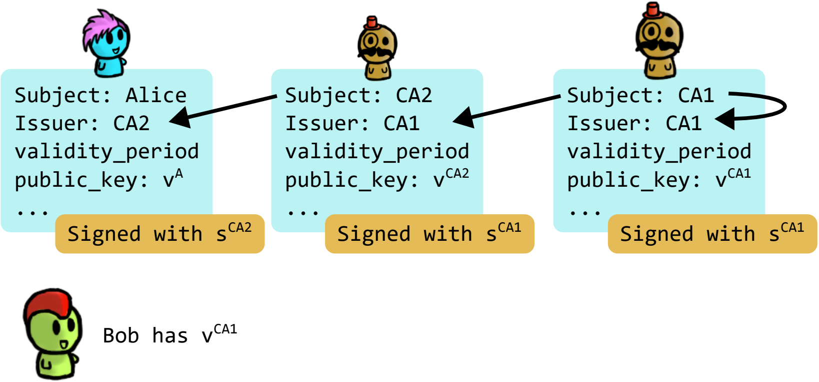
Certificate authorities

- Everyone is assumed to have a copy of the CA's verification key, so they can verify the signature on the certificate
- There can be multiple levels of CAs; level n CA issues certificates for level $n+1$ CAs—public-key infrastructure (PKI)
- Need to have only verification key of root CA to verify a certificate chain



Chain of certificates

Alice sends Bob the following certificate to prove her identity. Bob can follow the chain of certificates to validate Alice's identity.



Putting it all together

- We have all these blocks; now what?
- Put them together into protocols
- This is HARD. Just because your pieces all work, doesn't mean what you build out of them will; you have to *use* the pieces correctly: see a counterexample here.
- Common mistakes include:
  - Using the same stream cipher key for two messages
  - Assuming encryption also provides integrity
  - Falling for replay attacks or reaction attacks
  - *LOTS* more!