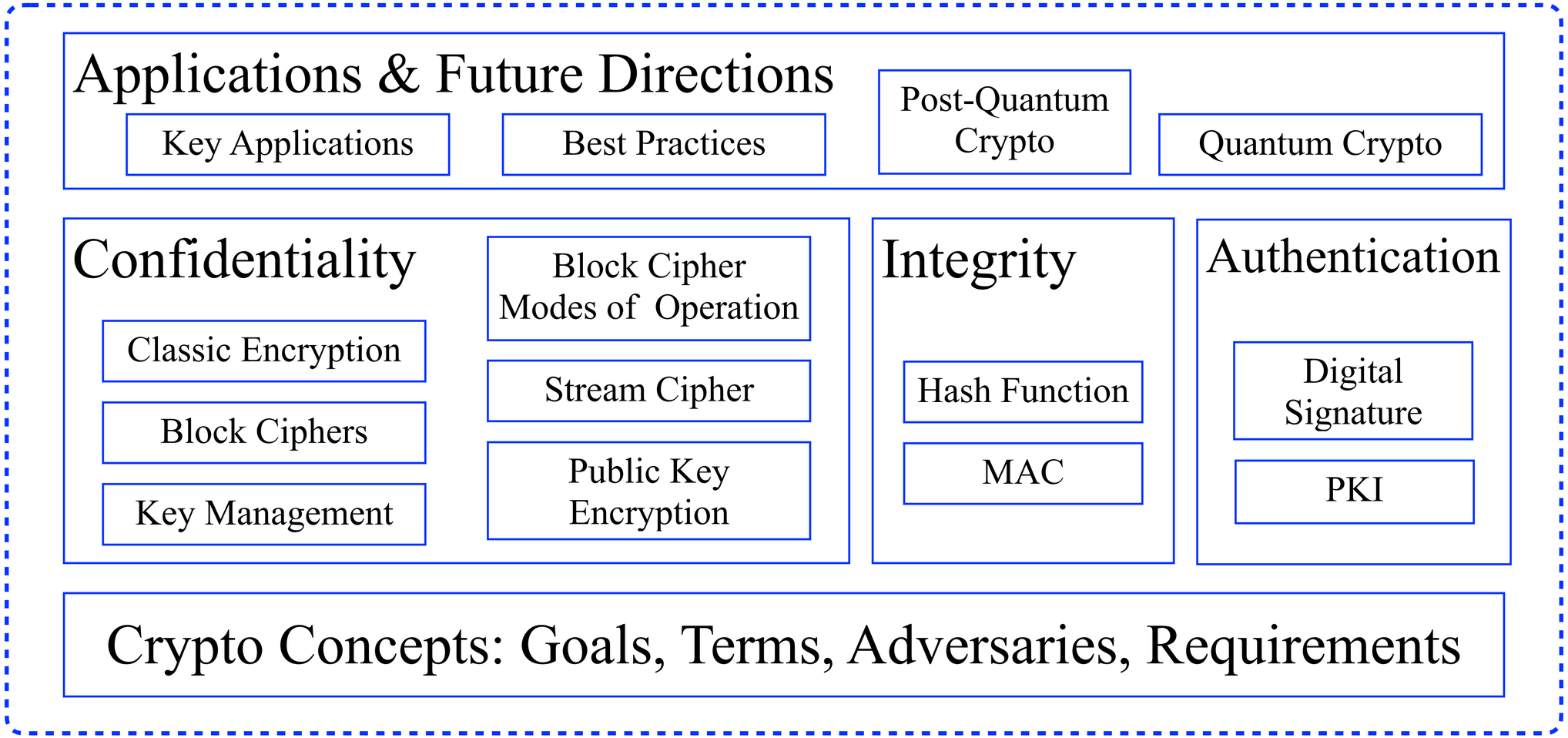
# Chapter 2: Cryptography

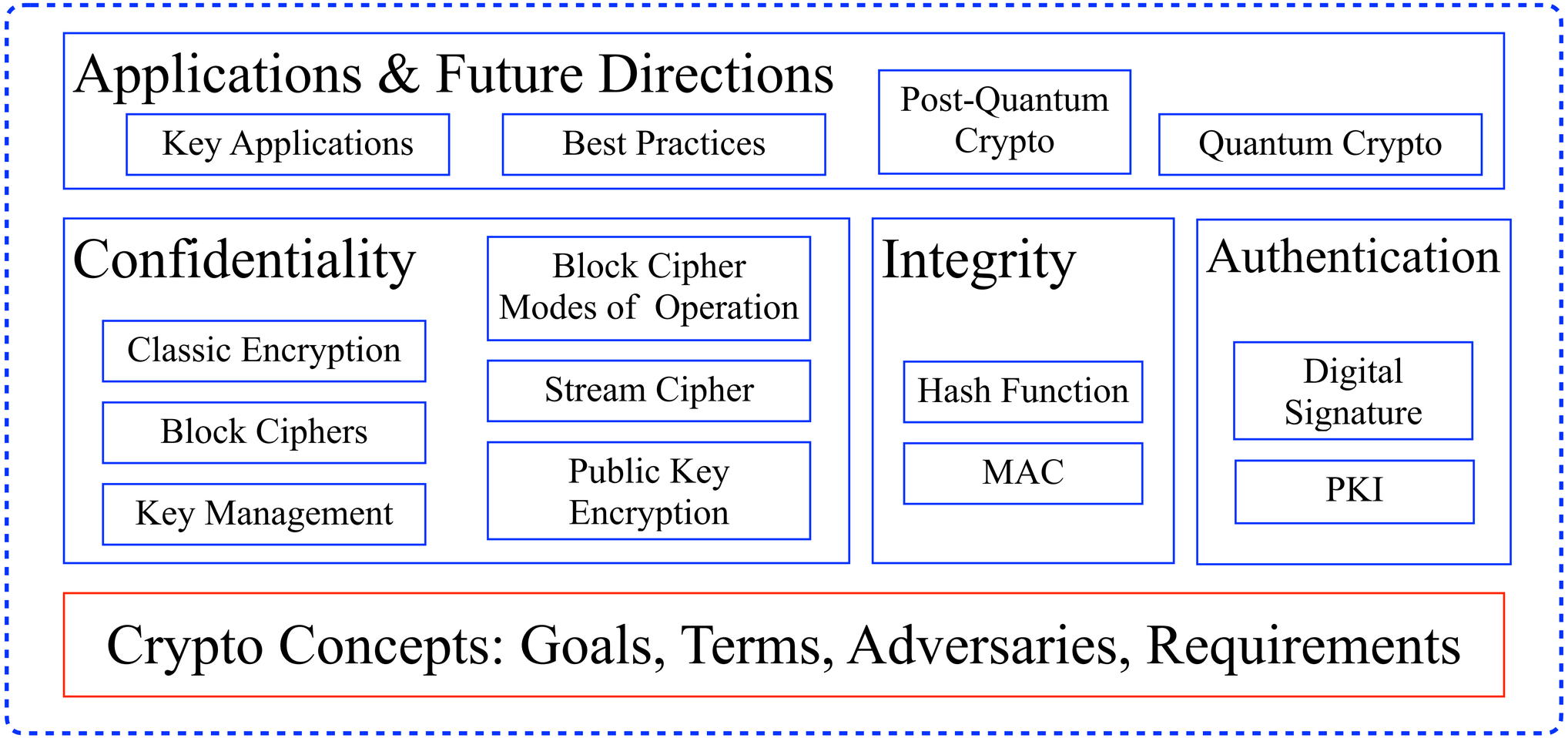
Xianghang Mi


中国科学技术大学
University of Science and Technology of China

# The Roadmap of Cryptography



# The Roadmap of Cryptography



# Crypto Concepts: What is Crypto?

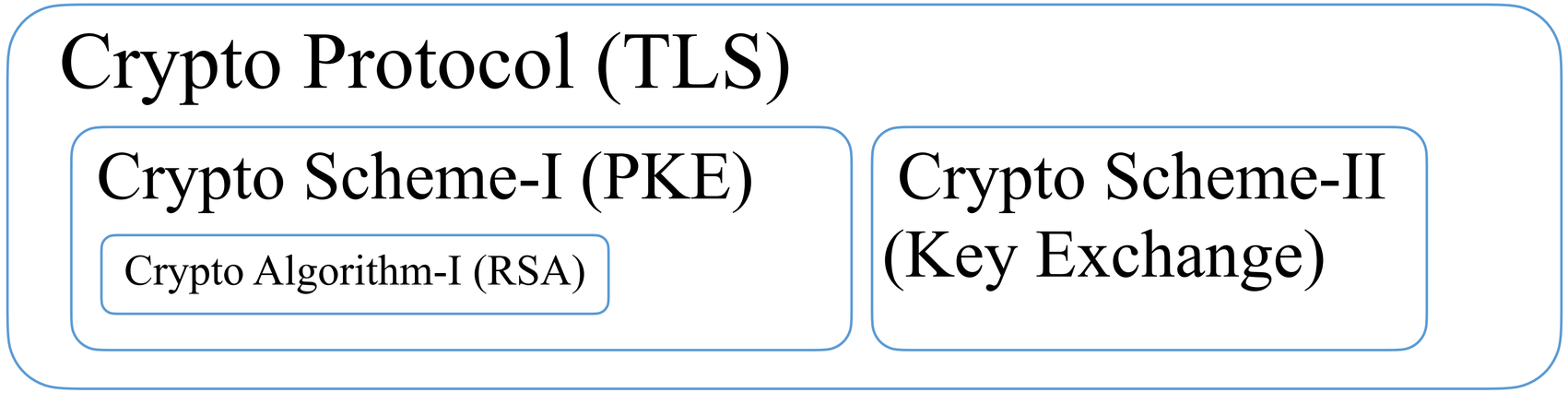
- Cryptography ≠ Encryption ≠ Password
  - Secure communication (transit) of data: confidentiality, integrity, authentication, non-repudiation, etc
  - Secure computing of data: privacy, anonymity, etc
- Crypto is a Mongrel
  - the intersection of mathematics, cs, information security, ee, physics, etc
- Crypto is a double-edged sword
  - Why?

# Crypto Concepts: Terms

- **Crypto Algorithm**: a well-defined procedure with specific inputs and outputs
  - Data Encryption Standard (DES)
  - RSA
- A Crypto **Scheme**: a collection of algorithms providing some functionality
  - The public key encryption (PKE) scheme

# Crypto Concepts: Terms

- A Crypto **Protocol**: a interactive system involving multiple parties
  - TLS
  - Secure messaging protocols



# Crypto Concepts: Symmetric Key vs Public Key Encryption

Symmetric Key Encryption:
The basic idea of symmetric key encryption is:

Message + Secret Key = Ciphertext

Ciphertext + Secret Key = Message

Both parties need **the same secret key** to encrypt and decrypt the message.

Public Key Encryption:
The basic idea of public key encryption is:

Message + Alice's Public Key = Ciphertext

Ciphertext + Alice's Private Key = Message

Anyone with Alice's public key can send Alice a secret message, but only Alice can decrypt.

# Crypto Concepts: Symmetric Key vs Public Key Encryption

Henceforth we denote a public/secret key pair (pk, sk), and a symmetric key by sk.

A message is denoted *m*, an encryption algorithm is denoted by Enc, a decryption algorithm by Dec, and a ciphertext by *c*.

For symmetric key schemes:

$\text{Enc}_{sk}(m) = c$ and $\text{Dec}_{sk}(c) = m$.

For public key schemes:

$\text{Enc}_{pk}(m) = c$ and $\text{Dec}_{sk}(c) = m$.

# Crypto Concepts: Digital Signature and MACs

## Digital Signature:

Message + Alice's Private Key = Signature
Message + Signature + Alice's Public Key = YES/NO

Alice can sign a message using her private key, and anyone can verify Alice's signature, since everyone can obtain her public key.

## MAC Functions:

Message + Secret Key = Tag
Message + Tag + Secret Key = YES/NO

Need the secret key to verify the tag.

# Crypto Concepts: Digital Signature and MACs

Henceforth we denote a public/secret key pair (pk, sk).

A message is denoted *m*, a signing algorithm is denoted Sig, a verification algorithm is denoted Verify, a signature is denoted *s*.

$\mathrm{Sig}_{sk}(m) = s$ and $\mathrm{Verify}_{pk}(s, m) =$ *YES*/*NO*.

In the case of MACs we have the tag production algorithm is MAC and the equations are

$\mathrm{MAC}_{sk}(m) = t$ and $\mathrm{Verify}_{sk}(t, m) =$ *YES*/*NO*.

# Crypto Concepts: Security Definitions

In much of cryptography security is defined by a game.
The game is between a Challenger and an Adversary.

The Adversary is given:

- A goal to achieve
- Powers it can use
- Restrictions on its operations

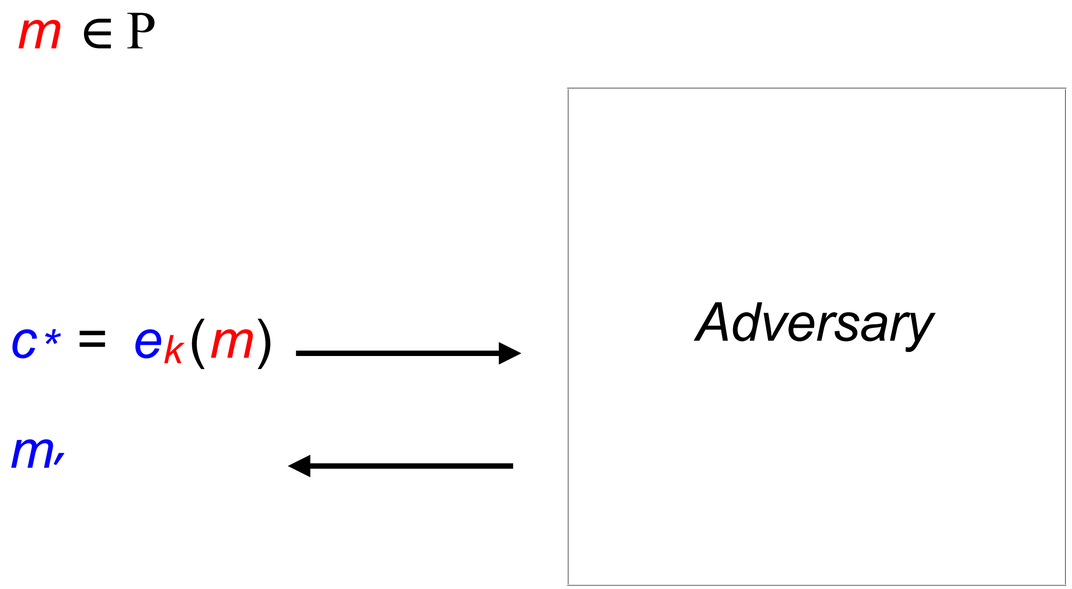
# Security Goals for Encryption

There are two main security goals

- OW: One way security. Can you decrypt a message?
- IND: Indistinguishability. Can you learn any information about a message?

# OW Security: Symmetric Key Case

Perhaps the most basic notion of security could be defined by the following game:



# IND-Security

This is the preferred security definition

Suppose that the challenger is given an encryption function $f$

- Defined by some key, i.e. $f(m) = e_k(m)$.

The attacker chooses two messages $m_1$ and $m_2$ of equal length.

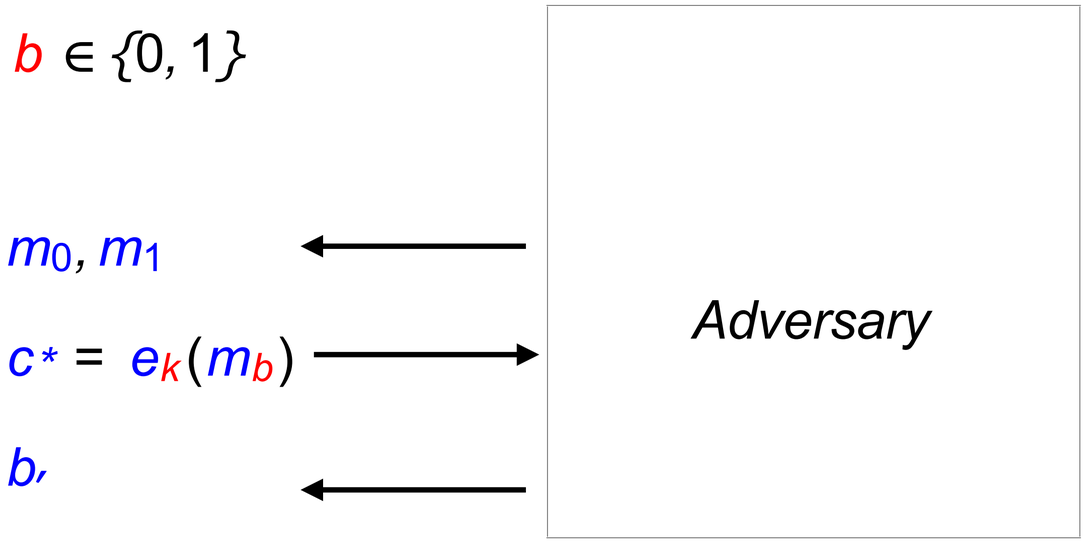
The challenger gives the attacker a ciphertext $c$ such that:

$c = f(m_1)$ or $c = f(m_2)$.

The goal is for the adversary to work out which message was encrypted.

# IND-Security: Symmetric Key Case

It is simpler to present this in terms of pictures representing a game played with the adversary *A*

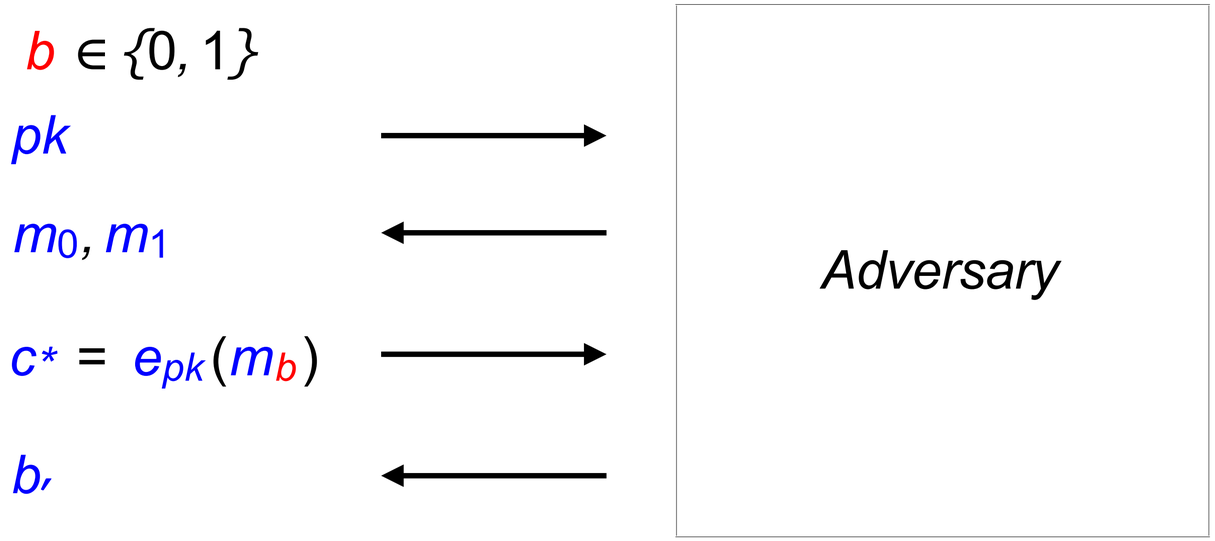



The ciphertext $c^*$ is called the target ciphertext.

Remember we must have $|m_0| = |m_1|$.

# IND-Security (Public Key Case)

For the public key case there is one main difference in the picture:



# Adversarial Powers

IND and OW are definitions of adversarial goals.

- They say nothing about what powers we give the adversary

We define powers by giving the adversary access to various oracles.

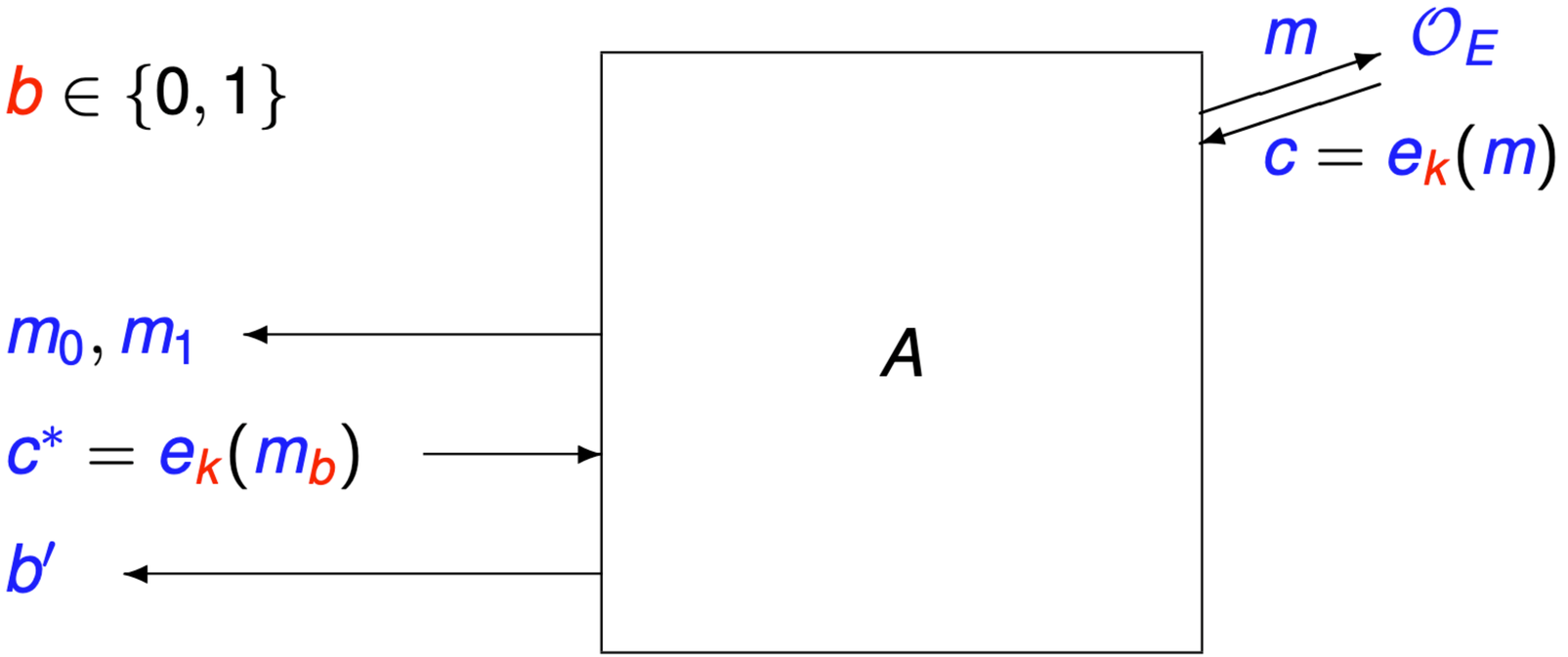
Passive Attack
The adversary is given no oracles

Chosen Plaintext Attack (CPA)
The adversary can encrypt any message of his choosing.

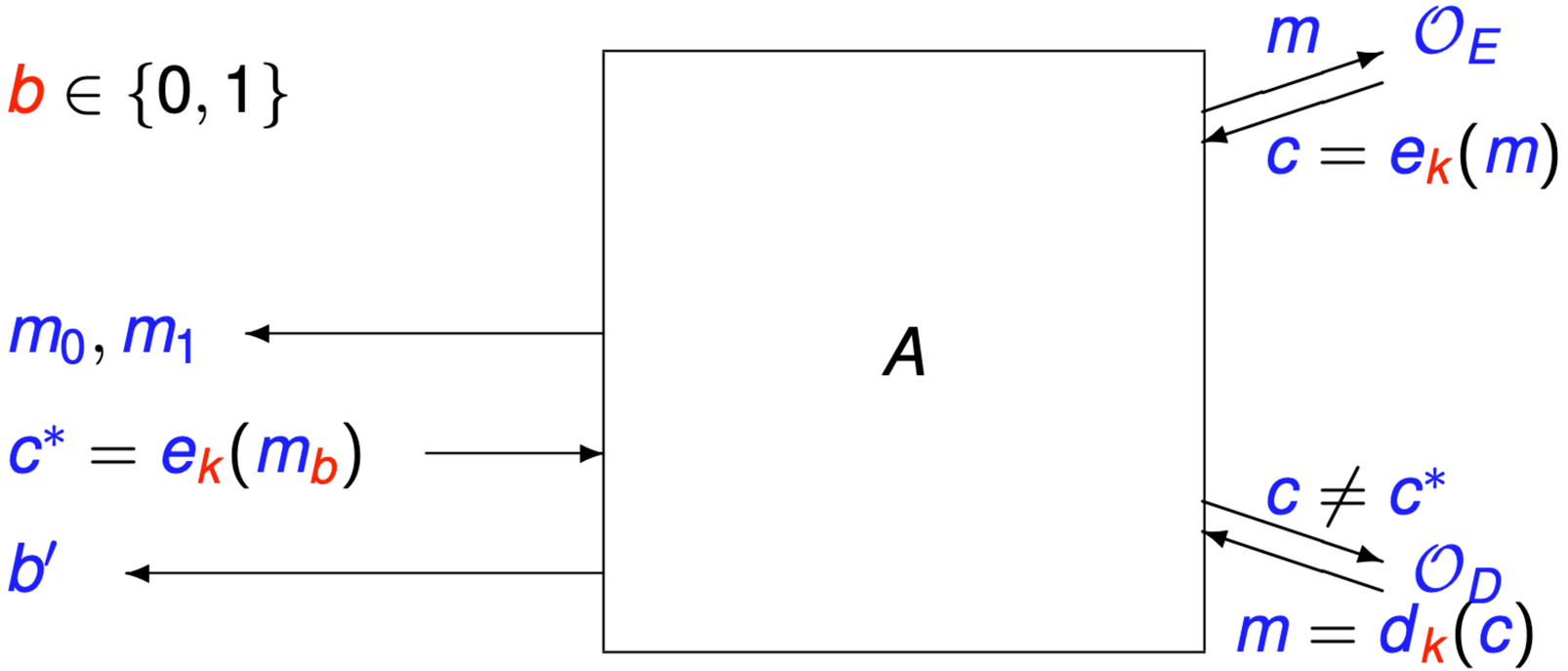
Chosen Ciphertext Attack (CCA)
The adversary can decrypt any message of his choosing, except he is not allowed to decrypt $c^*$.

We say a scheme is IND-PASS, IND-CPA, IND-CCA, OW-PASS, OW-CPA, OW-CCA etc.

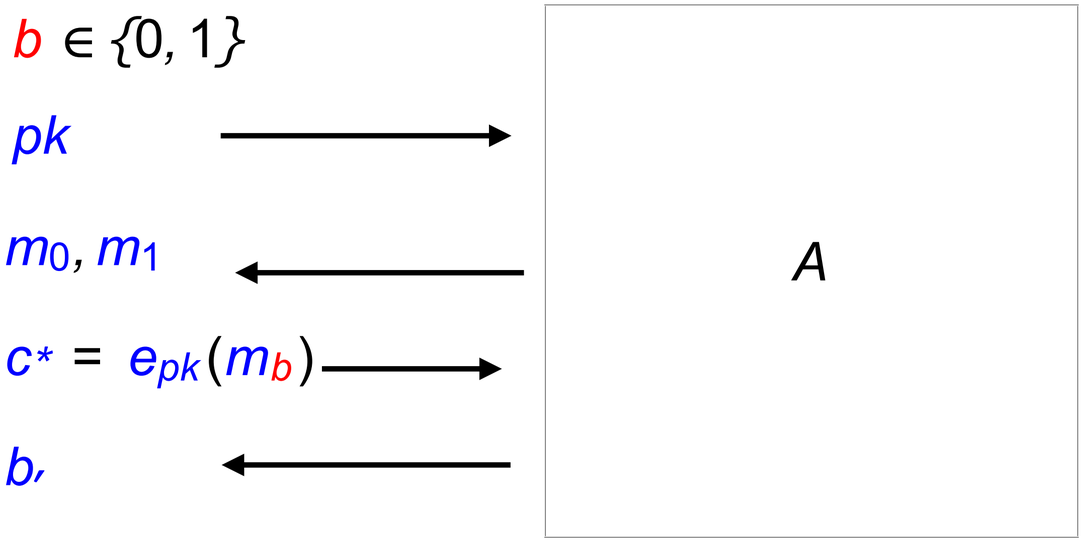
# IND-CPA Symmetric Case



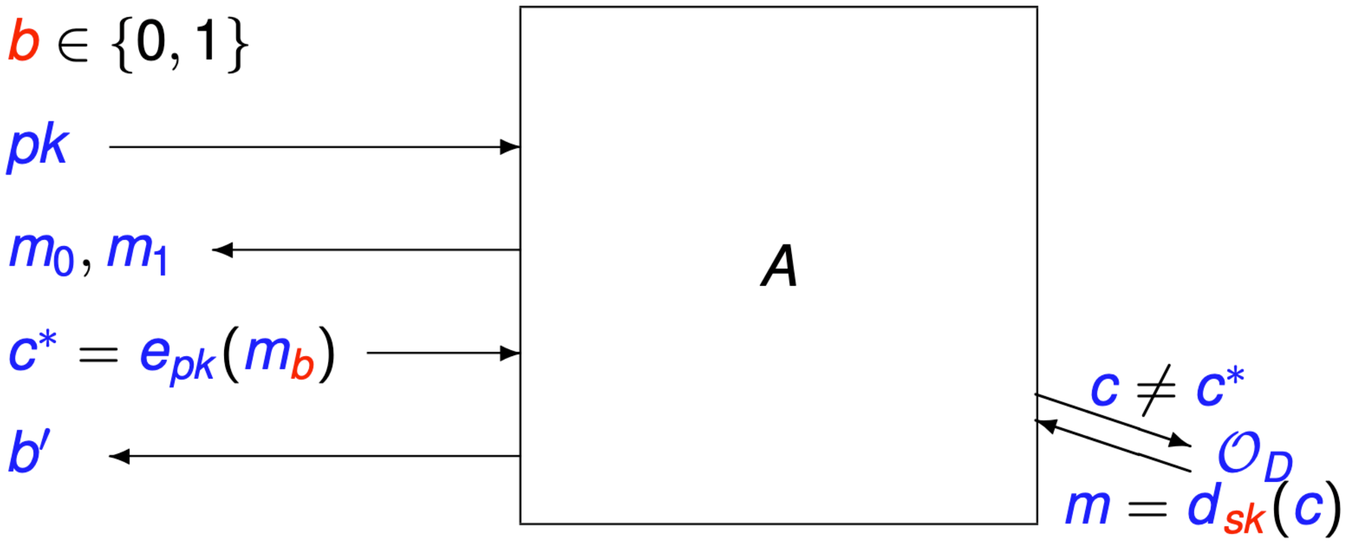
# IND-CCA Symmetric Case



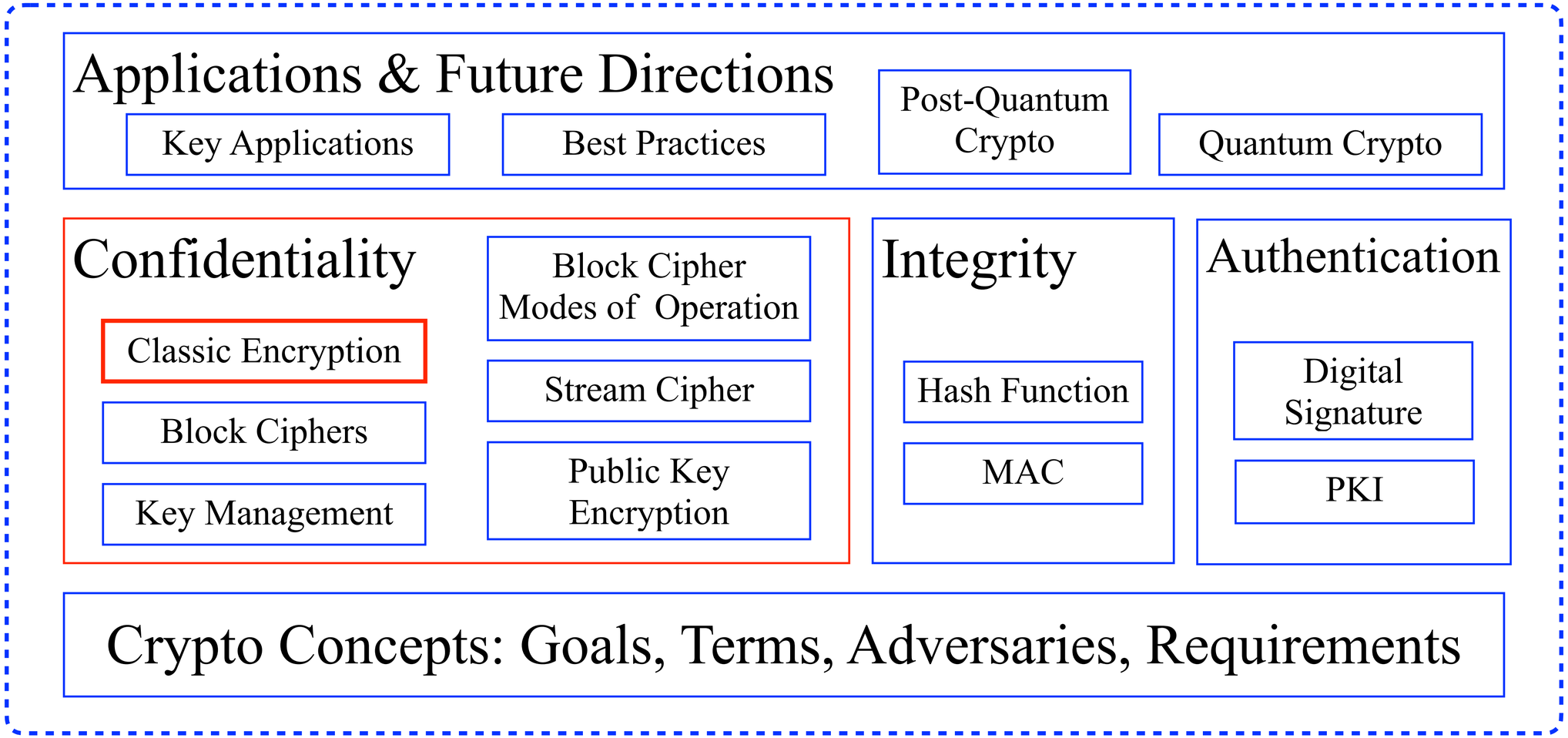
# IND-CPA Public Key Case



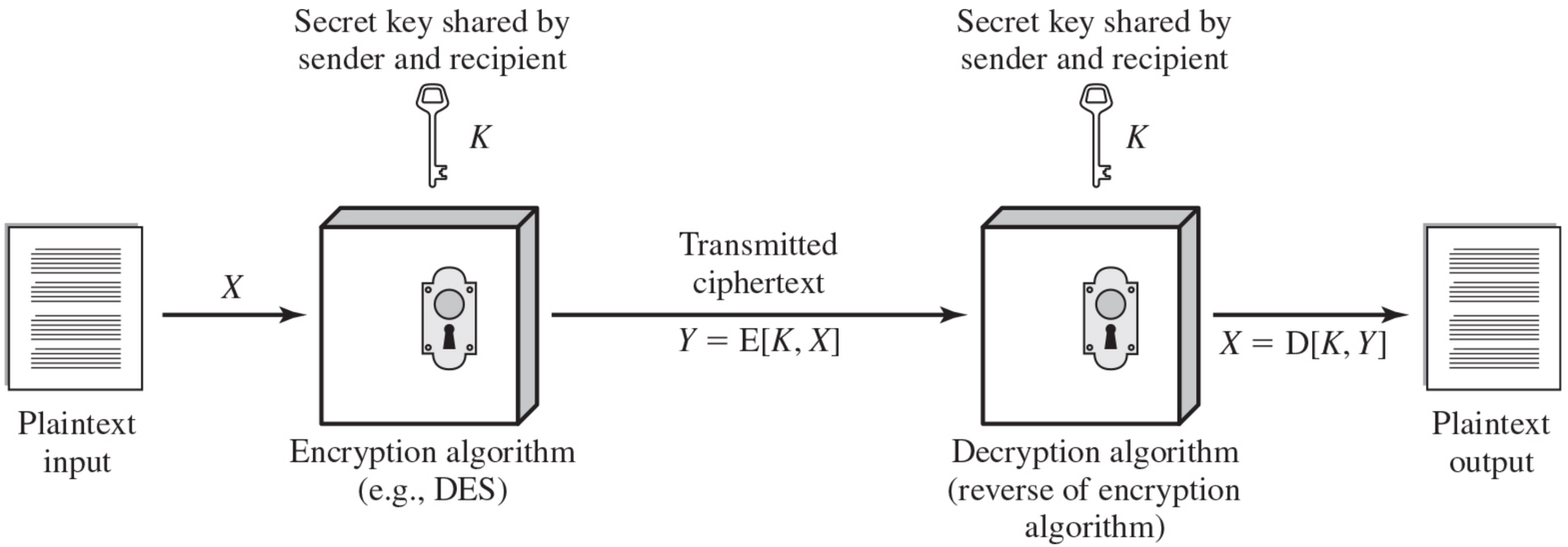
# IND-CCA Public Key Case



# The Roadmap of Cryptography



# Symmetric Encryption

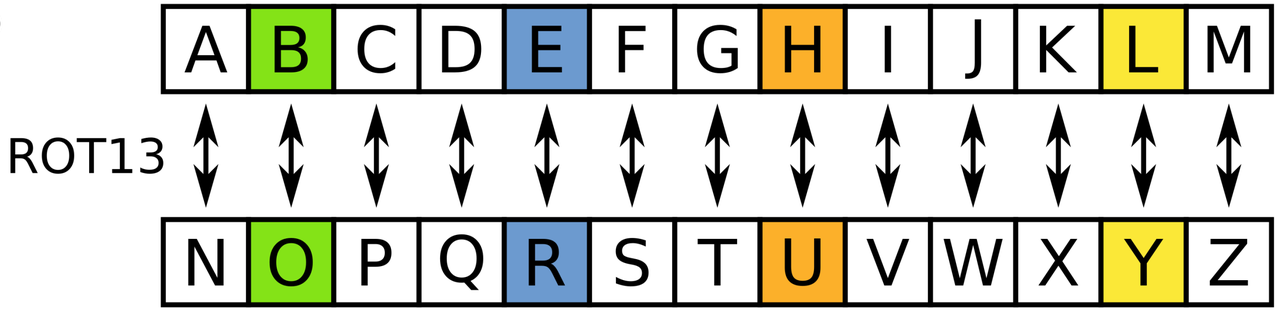


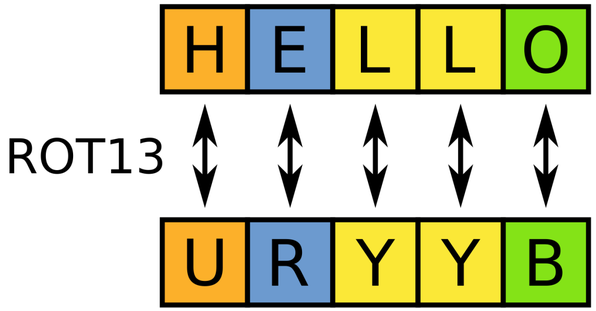
# Symmetric Encryption

- Classic ciphers
  - Substitution ciphers
  - Transposition ciphers
  - Concealment ciphers
- Block ciphers
  - DES
  - AES
- Stream ciphers

# Substitution Ciphers

- ROT13: a letter substitution cipher, a special form of the Ceasar cipher
- Ceasar cipher: given shift value $x$, letter $l_i$ is replaced with $l_{(i+x)\%26}$



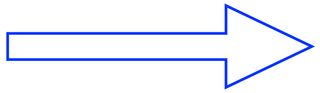
# Substitution Ciphers

- Encryption: units in the plaintext are replaced with units in the ciphertext, following a defined manner
- Decryption: the reverse substitution
- The unit can be a letter, or a pair of letters, etc

# Transposition ciphers

- Encryption: reorder units of the plaintext following a system

Welcome to this course!



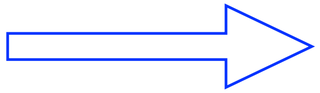
coce ltss eeir oto! Wmhu

| 5 | 3 | 2 | 1 | 4 |
|---|---|---|---|---|
| W | e | l | c | o |
| m | e | t | o | t |
| h | i | s | c | o |
| u | r | s | e | ! |

# Transposition ciphers

- Encryption: reorder units of the plaintext following a system

Welcome to this course!



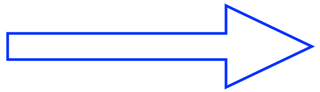
coce ltss eeir oto! Wmhu

| 5 | 3 | 2 | 1 | 4 |
|---|---|---|---|---|
| W | e | l | c | o |
| m | e | t | o | t |
| h | i | s | c | o |
| u | r | s | e | ! |

# Transposition ciphers

- Encryption: reorder units of the plaintext following a system

Welcome to this course!



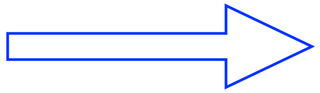
coce ltss eeir oto! Wmhu

| 5 | 3 | 2 | 1 | 4 |
|---|---|---|---|---|
| W | e | l | c | o |
| m | e | t | o | t |
| h | i | s | c | o |
| u | r | s | e | ! |

# Transposition ciphers

- Route cipher

Welcome to this course!

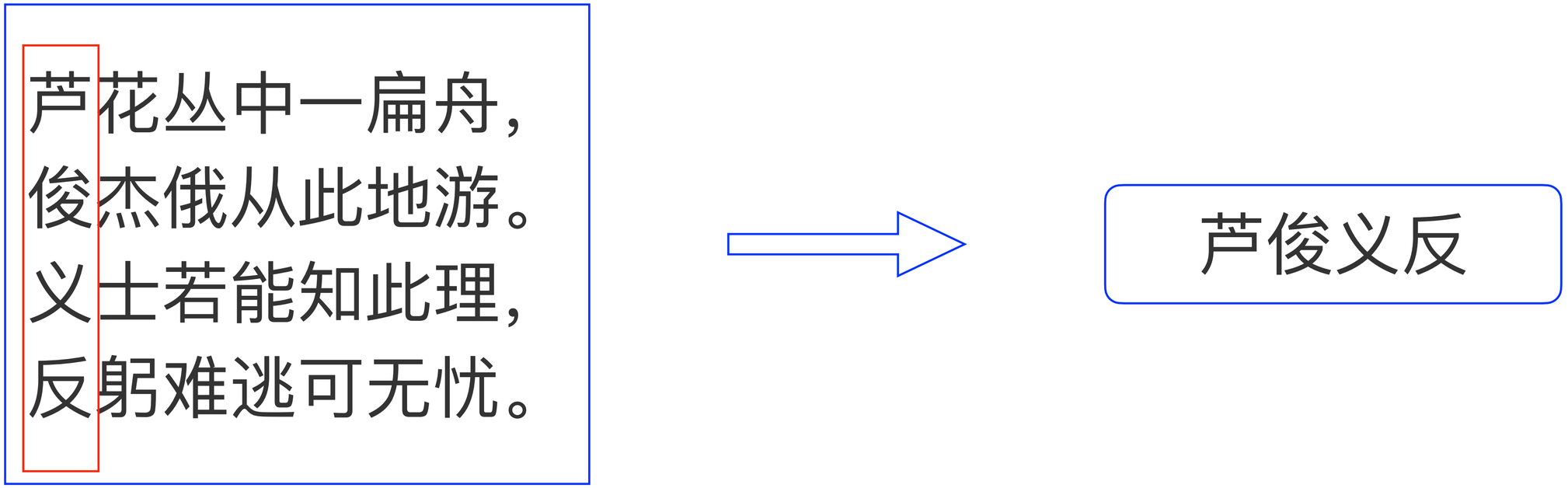


oto! esru hmWelc ocsiet

| spiral inwards, clockwise, starting from the top right | | | | |
|---|---|---|---|---|
| W | e | l | c | o |
| m | e | t | o | t |
| h | i | s | c | o |
| u | r | s | e | ! |

# Concealment (null) ciphers

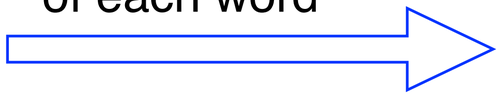
- Encryption: plaintext units are mixed with a large amount of non-cipher material
- A simple form of steganography



# Concealment (null) ciphers

PRESIDENT'S EMBARGO RULING SHOULD HAVE IMMEDIATE NOTICE. GRAVE SITUATION AFFECTING INTERNATIONAL LAW. STATEMENT FORESHADOWS RUIN OF MANY NEUTRALS. YELLOW JOURNALS UNIFYING NATIONAL EXCITEMENT IMMENSELY.

Get the first letter of each word



Pershing sails from N.Y. June I

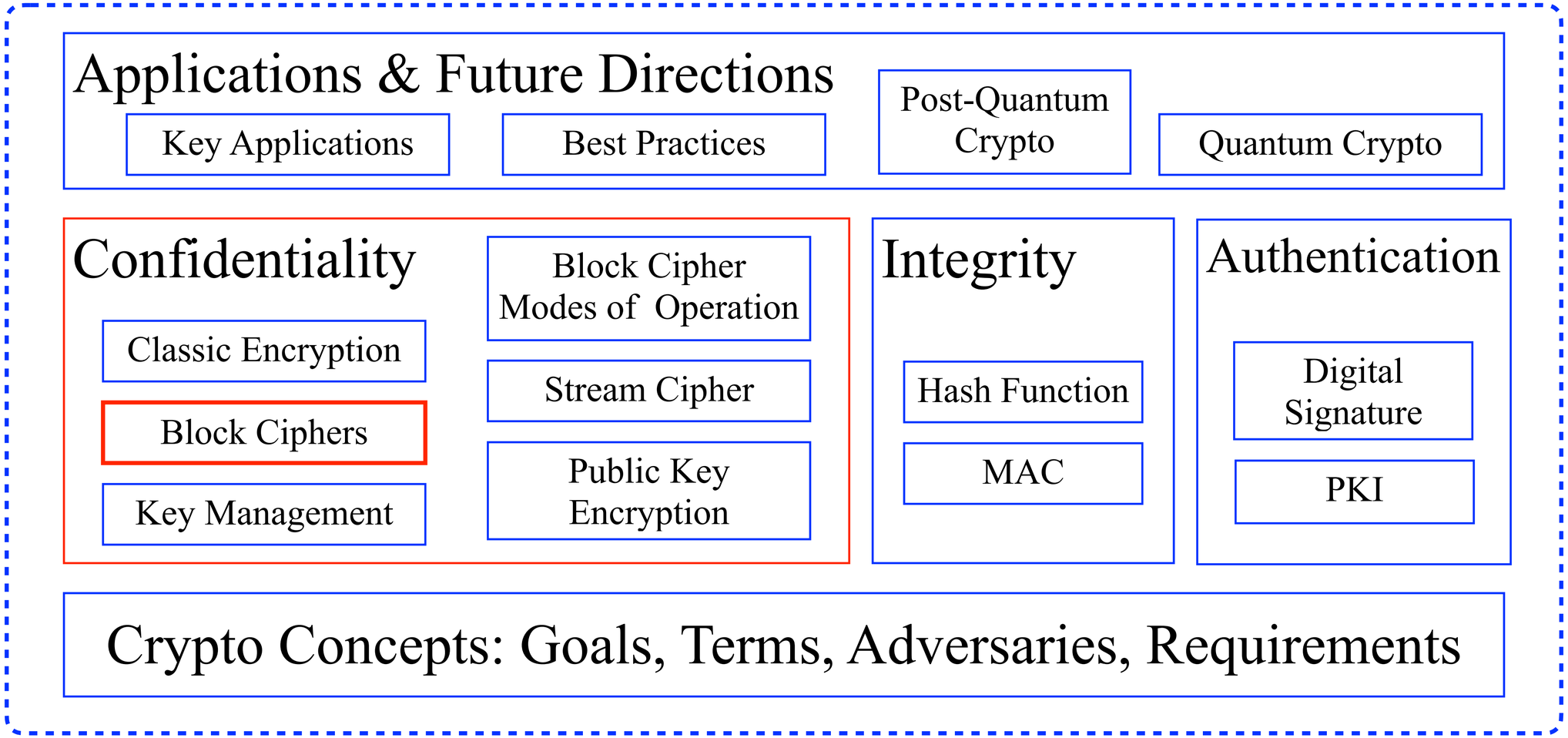
# Classic ciphers

- Given a text message, multiple classic ciphers can be jointly applied
- Most can be practically computed and solved by hand, which is different from block ciphers and stream ciphers
- Vulnerable: chosen plaintext attacks, chosen ciphertext attacks, frequency distribution analysis, or even passive attacks

# Classic ciphers

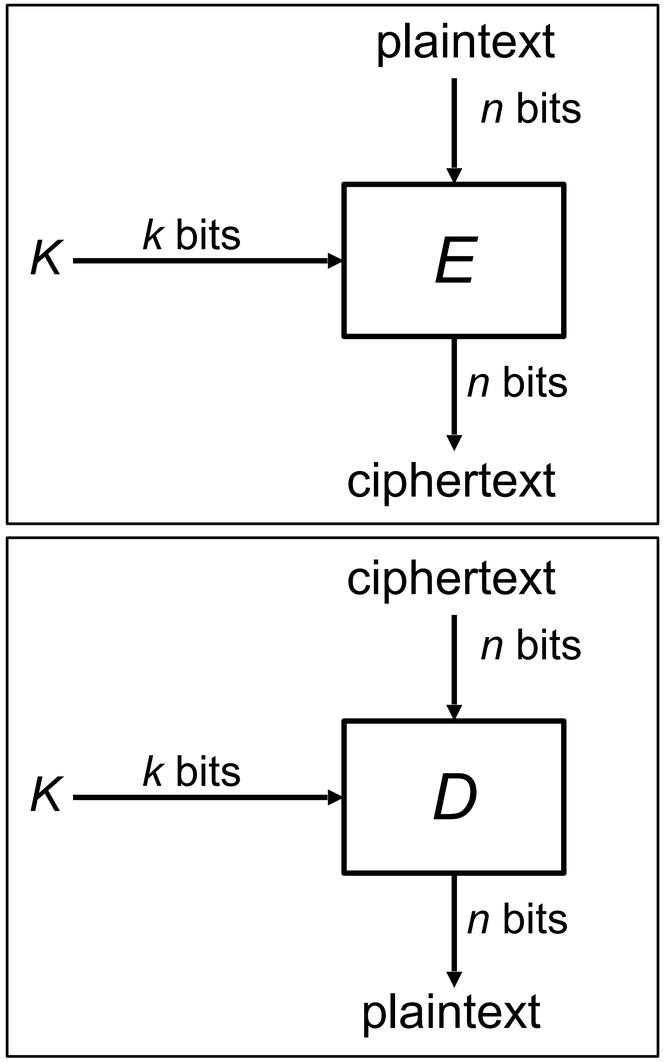
- Most cannot fit in well in modern computing and communication
  - Data are transmitted or stored as either blocks of bits or streams of bits, rather than characters
  - The adversaries have much more power in terms of breaking the encryption

# The Roadmap of Cryptography



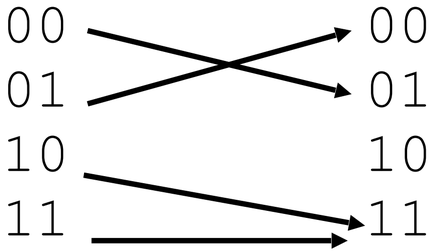
# Block Ciphers: Definition

- **Block cipher**: An encryption/decryption algorithm that encrypts a fixed-sized block of bits
- $E_K(M) \rightarrow C$: Encryption
  - Inputs: $k$-bit key $K$ and an $n$-bit plaintext $M$
  - Output: An $n$-bit ciphertext $C$
  - Sometimes written as: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$
- $D_K(C) \rightarrow M$: Decryption
  - Inputs: a $k$-bit key, and an $n$-bit ciphertext $C$
  - Output: An $n$-bit plaintext
  - Sometimes written as: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$
  - The inverse of the encryption function
- Properties
  - **Correctness**: $E_K$ is a permutation, $D_K$ is its inverse
  - **Efficiency**: Encryption/decryption should be fast
  - **Security**: $E$ behaves like a random permutation

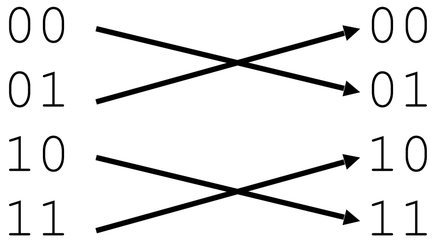


# Block Ciphers: Correctness

- $E_K(M)$ must be a **permutation** (**bijective function**) on $n$-bit strings
  - Each input must correspond to exactly one unique output
- Intuition
  - Suppose $E_K(M)$ is not bijective
  - Then two inputs might correspond to the same output: $E(K, x_1) = E(K, x_2) = y$
  - Given ciphertext $y$, you can't uniquely decrypt. $D(K, y) = x_1$? $D(K, y) = x_2$?




Not bijective: Two inputs encrypt to the same output




Bijective: Each input maps to exactly one unique output

# Block Ciphers: Security

- A secure block cipher behaves like a randomly chosen permutation from the set of all permutations on *n*-bit strings
  - A random permutation: Each *n*-bit input is mapped to one randomly-chosen *n*-bit output
- Defined by a distinguishing game
  - Eve gets two boxes: One is a randomly chosen permutation, and one is $E_K$ with a randomly chosen key *K*
  - Eve should not be able to tell which is which with probability > 1/2


One of these is $E_K$ with a randomly chosen *K*, and the other one is a randomly chosen permutation. Eve can't distinguish them.

# Block ciphers: Brute-force attacks?

- How hard is it to run a brute-force attack on a 128-bit key?
  - We have to try $2^{128}$ possibilities. How big is $2^{128}$?
- Handy approximation: $2^{10} \approx 10^3$
  - $2^{128} = 2^{10*12.8} \approx (10^3)^{12.8} \approx (10^3)^{13} = 10^{39}$
- Suppose we have massive hardware that can try $10^9$ (1 billion) keys in 1 nanosecond (a billionth of a second). That's $10^{18}$ keys per second
  - We'll need $10^{39} / 10^{18} = 10^{21}$ seconds. How long is that?
  - One year $\approx 3 \times 10^7$ seconds
  - $10^{21}$ seconds / $3 \times 10^7 \approx 3 \times 10^{13}$ years $\approx$ 30 trillion years
- **Takeaway**: Brute-forcing a 128-bit key takes astronomically long. Don't even try.

# Block Ciphers: Efficiency

- Encryption and decryption should be computable in microseconds
  - Formally: KeyGen(), Enc(), and Dec(), should not take exponential time
- Block cipher algorithms typically use operations like XOR, bit-shifting, and small table lookups
  - Very fast on modern processors
- Modern CPUs provide dedicated hardware support for block ciphers

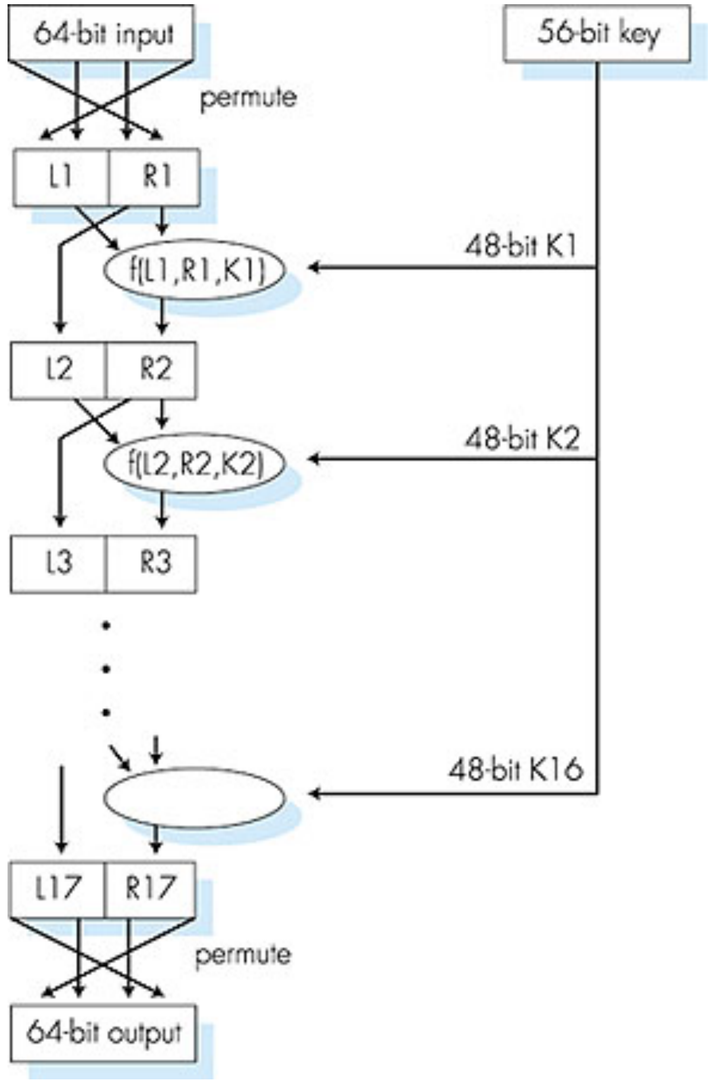
# DES (Data Encryption Standard)

- Designed in late 1970s
- Block size 64 bits ($n = 64$)
- Key size 56 bits ($k = 56$)
- NSA influenced two facets of its design
  - Altered some subtle internal workings in a mysterious way
  - Reduced key size from 64 bits to 56 bits
  - Made brute force attacks feasible for an attacker with massive computational resources (by 1970s standards)
- The algorithm remains essentially unbroken 40 years later
  - The NSA's tweaking hardened it against an attack publicly revealed a decade later

- **Insecure now**: In 2006, a FPGA-based parallel machine broke DES in 9 days at a $10,000 hardware cost.

# DES

- Initial permutation
- 16 identical "rounds"
- In each round, applying the Feistel (F) function to half of the input



# AES (Advanced Encryption Standard)

- 1997–2000: NIST (National Institute of Standards and Technology) in the US held a competition to pick a new block cipher standard
- Out of the 5 finalists:
  - Rijndael, Twofish, and Serpent had really good performance
  - RC6 had okay performance
  - Mars had ugly performance
- On any given computing platform, Rijndael was *never* the fastest
- But on every computing platform, Rijndael was *always* the second-fastest
  - Twofish and Serpent each had at least one compute platform they were bad at
- Rijndael was selected as the new block cipher standard

# AES (Advanced Encryption Standard)

- Key size 128, 192, or 256 bits ($k$ = 128, 192, or 256)
  - Actual cipher names are AES-128, AES-192, and AES-256
  - Paranoid people like the NSA use AES-256 keys, but AES-128 is just fine in practice
- Block size 128 bits ($n$ = 128)
  - Note: The block size is still always 128 bits, regardless of key size
- You don’t need to know how AES works, but you do need to know its parameters

# AES (Advanced Encryption Standard)

- There is no formal proof that AES is secure (indistinguishable from a random permutation)
- However, in 20 years, nobody has been able to break it, so it is *assumed* to be secure
  - The NSA uses AES-256 for secrets they want to keep secure for the 40 years (even in the face of unknown breakthroughs in research)
- **Takeaway**: AES is the modern standard block cipher algorithm
  - The standard key size (128 bits) is large enough to prevent brute-force attacks

# A Comparison of Block Ciphers

| | DES | Triple DES | AES |
|---|---|---|---|
| **Plaintext block size (bits)** | 64 | 64 | 128 |
| **Ciphertext block size (bits)** | 64 | 64 | 128 |
| **Key size (bits)** | 56 | 112 or 168 | 128, 192, or 256 |

DES = Data Encryption Standard

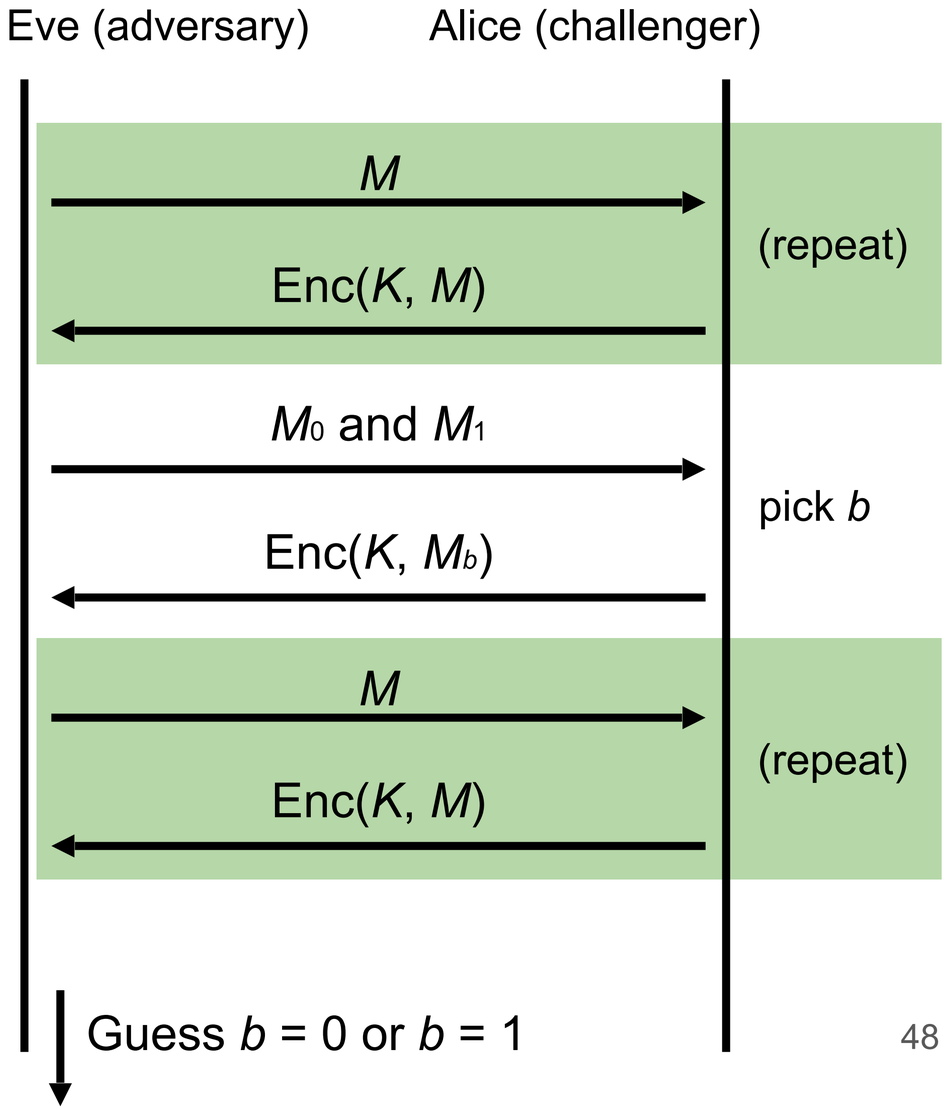
AES = Advanced Encryption Standard

# A Comparison of Block Ciphers

| Key Size (bits) | Cipher | Number of Alternative Keys | Time Required at $10^9$ decryptions / $\mu$s | Time Required at $10^{13}$ decryptions / $\mu$s |
|---|---|---|---|---|
| 56 | DES | $2^{56} \approx 7.2 \times 10^{16}$ | $2^{55}\ \mu s = 1.125$ years | 1 hour |
| 128 | AES | $2^{128} \approx 3.4 \times 10^{38}$ | $2^{127}\ \mu s = 5.3 \times 10^{21}$ years | $5.3 \times 10^{17}$ years |
| 168 | Triple DES | $2^{168} \approx 3.7 \times 10^{50}$ | $2^{167}\ \mu s = 5.8 \times 10^{33}$ years | $5.8 \times 10^{29}$ years |
| 192 | AES | $2^{192} \approx 6.3 \times 10^{57}$ | $2^{191}\ \mu s = 9.8 \times 10^{40}$ years | $9.8 \times 10^{36}$ years |
| 256 | AES | $2^{256} \approx 1.2 \times 10^{77}$ | $2^{255}\ \mu s = 1.8 \times 10^{60}$ years | $1.8 \times 10^{56}$ years |

# Are Block Ciphers IND-CPA Secure?

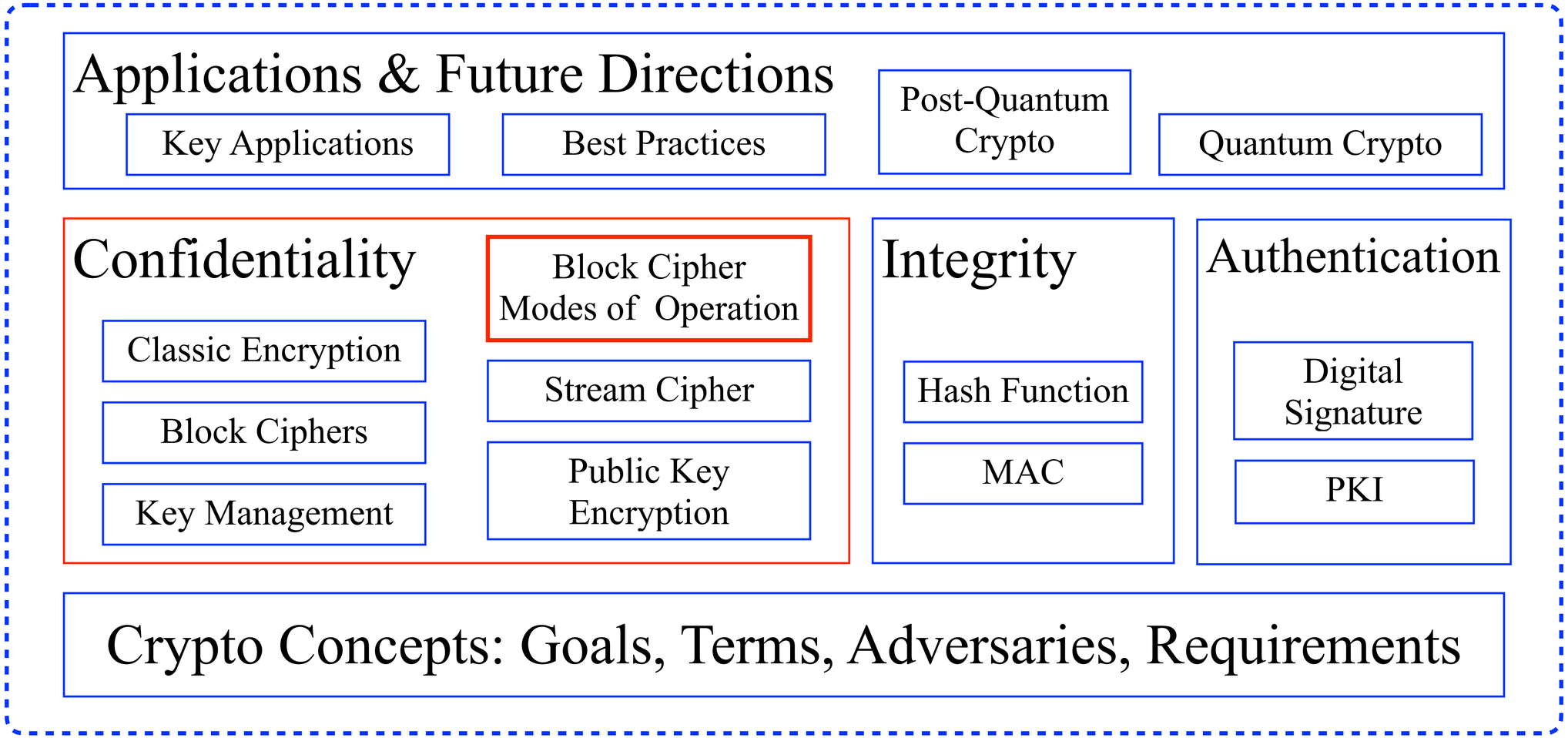
- Consider the following adversary:
  - Eve sends two different messages $M_0$ and $M_1$
  - Eve receives either $E_K(M_0)$ or $E_K(M_1)$
  - Eve requests the encryption of $M_0$ again
  - Strategy: If the encryption of $M_0$ matches what she received, guess b = 0. Else, guess b = 1.
- Eve can win the IND-CPA game with probability 1!
  - Block ciphers are not IND-CPA secure



# Issues with Block Ciphers

- Block ciphers are not IND-CPA secure, because they're deterministic
  - A scheme is **deterministic** if the same input always produces the same output
  - No deterministic scheme can be IND-CPA secure because the adversary can always tell if the same message was encrypted twice
- Block ciphers can only encrypt messages of a fixed size
  - For example, AES can only encrypt-decrypt 128-bit messages
  - What if we want to encrypt something longer than 128 bits?
- To address these problems, we'll add **modes of operation** that use block ciphers as a building block!

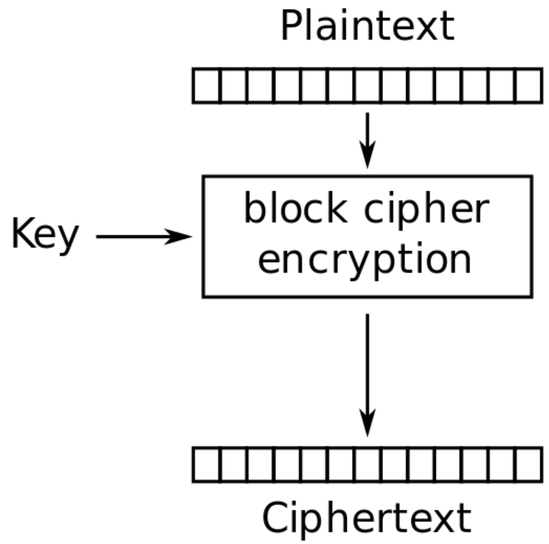
# The Roadmap of Cryptography



# Scratchpad: Let's design it together

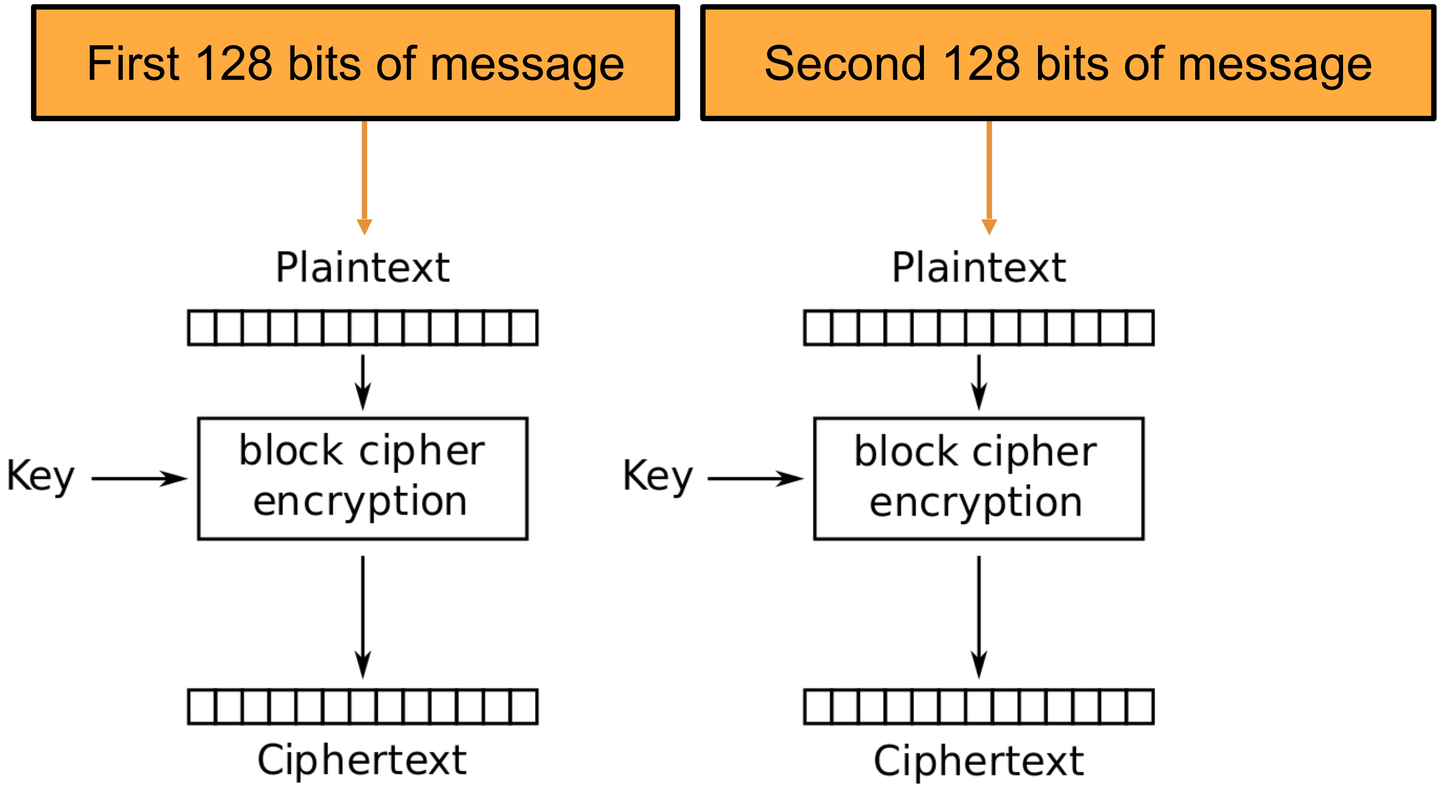
Here's an AES block. Remember that it can only encrypt 128-bit messages.

How can we use AES to encrypt a longer message (say, 256 bits?)



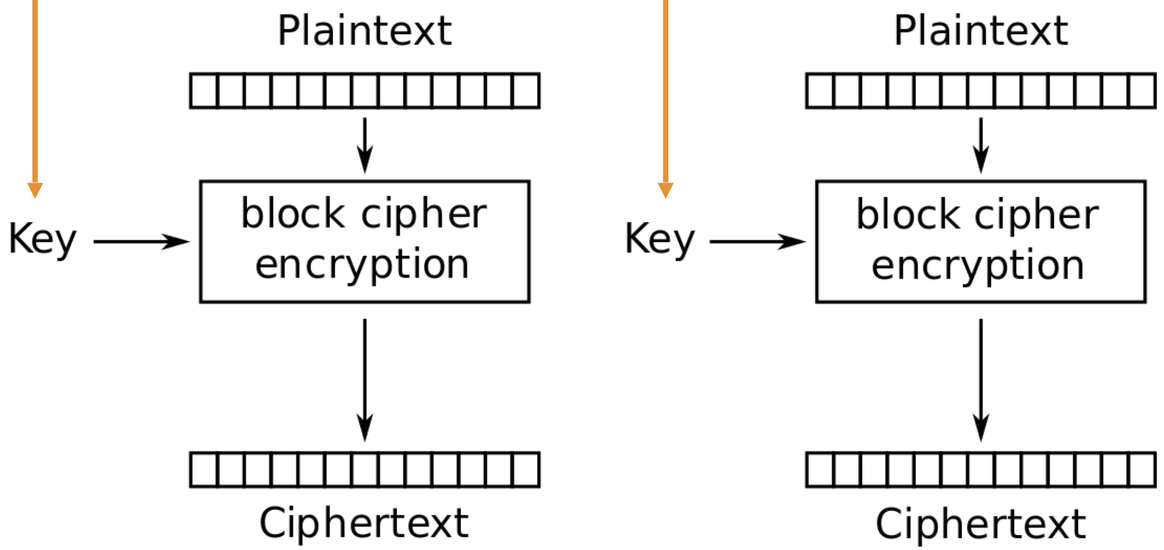
# Scratchpad: Let's design it together

Idea: Let's use AES twice!



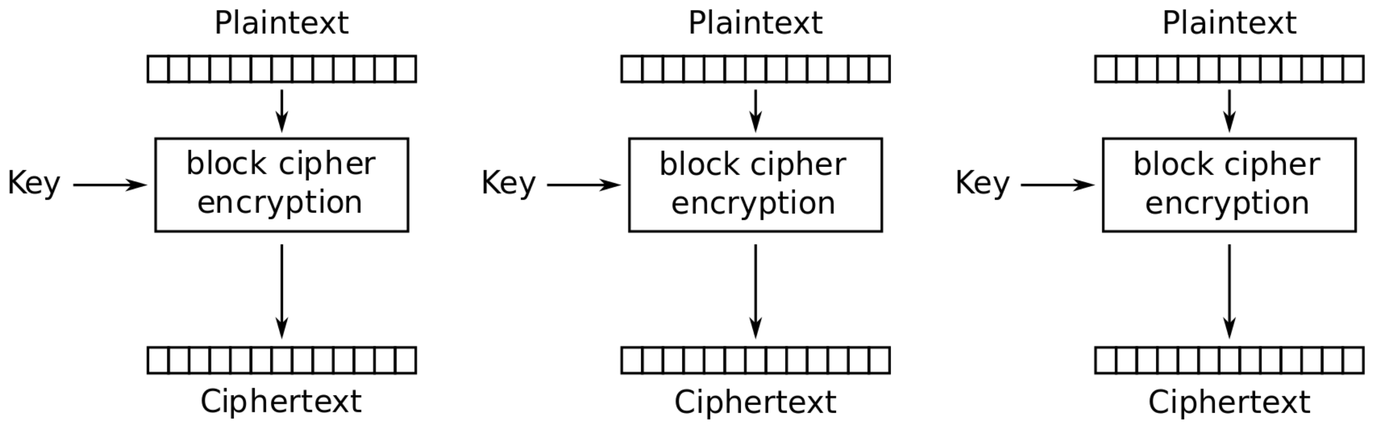
# Scratchpad: Let's design it together

Note that we are using the same key twice. We want to avoid a situation like one-time pads where we need very long keys.



# ECB Mode

- We've just designed **electronic code book (ECB) mode**
  - Enc($K$, $M$) = $C_1$ || $C_2$ || … || C$m$
  - Assume m is the number of blocks of plaintext in $M$, each of size $n$
- AES-ECB is not IND-CPA secure. Why?
  - Because ECB is deterministic

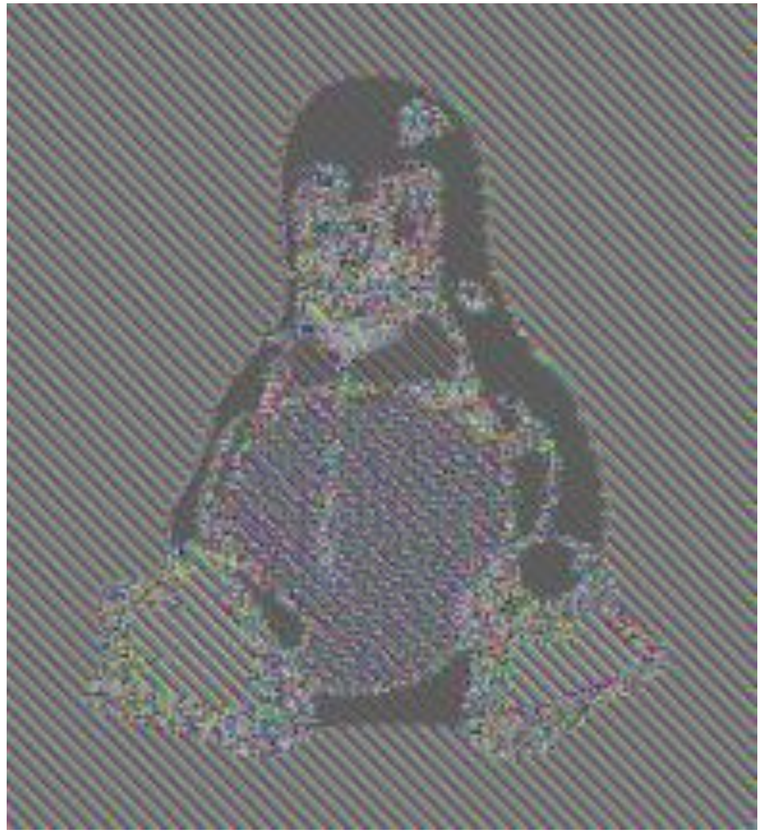



Electronic Codebook (ECB) mode encryption

# ECB Mode: Penguin

Original image

# ECB Mode: Penguin

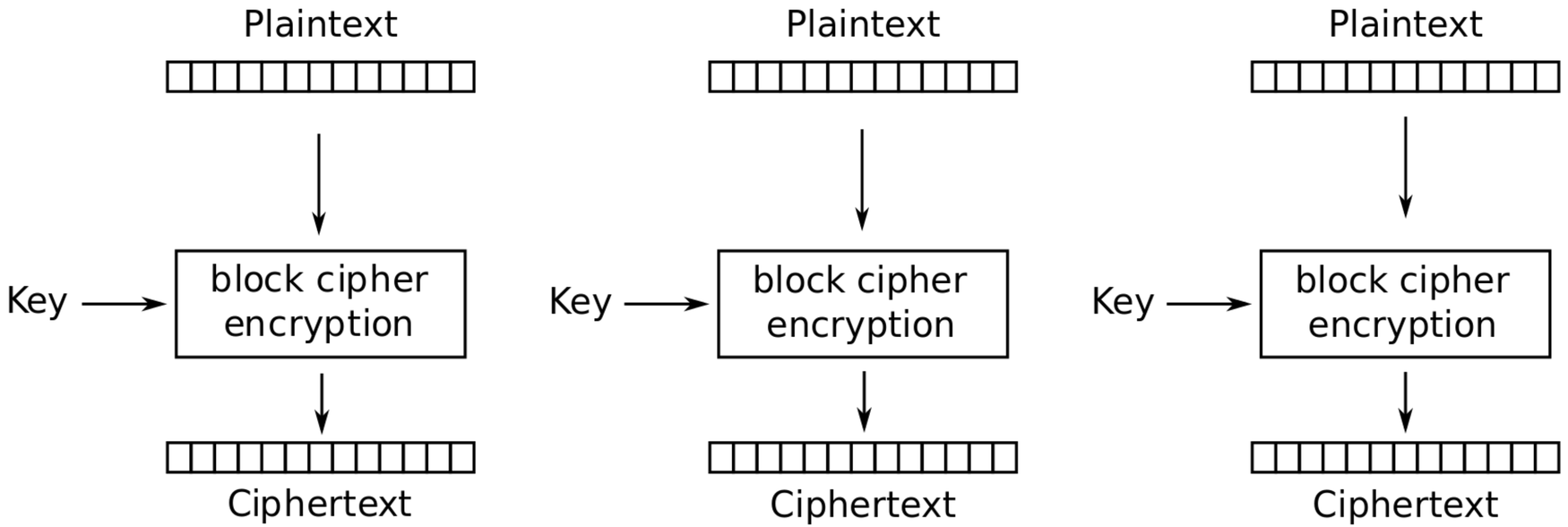


Encrypted with ECB

# Scratchpad: Let's design it together

Here's ECB mode. It's not IND-CPA secure because it's deterministic.

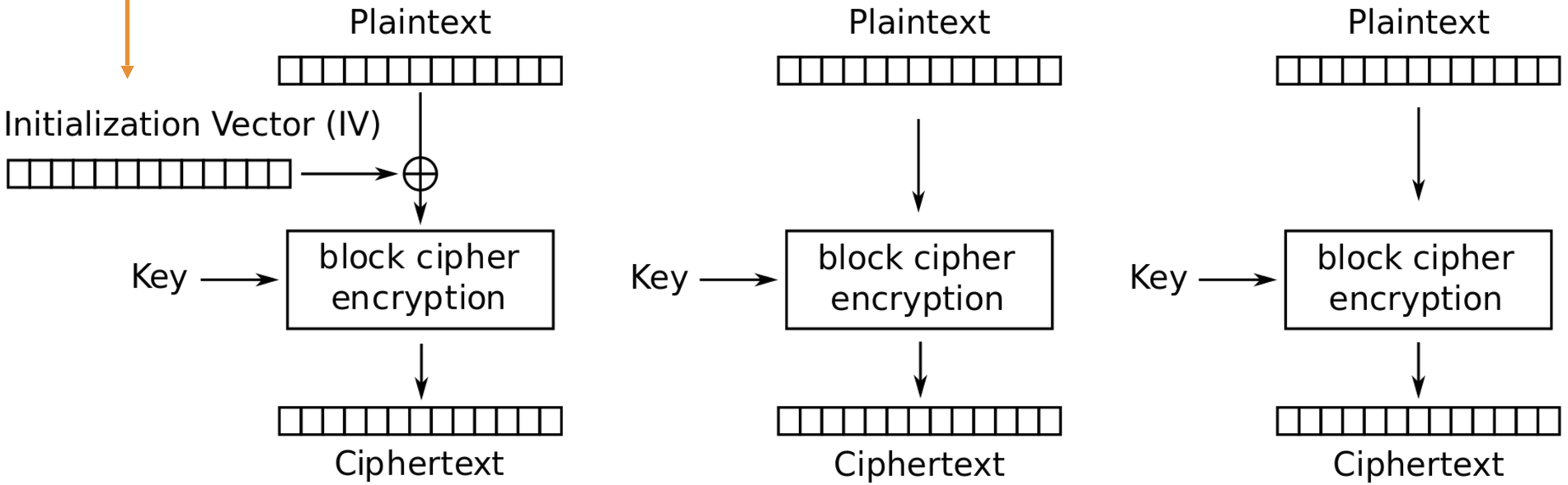
Let's fix that by adding some randomness.



# Scratchpad: Let's design it together

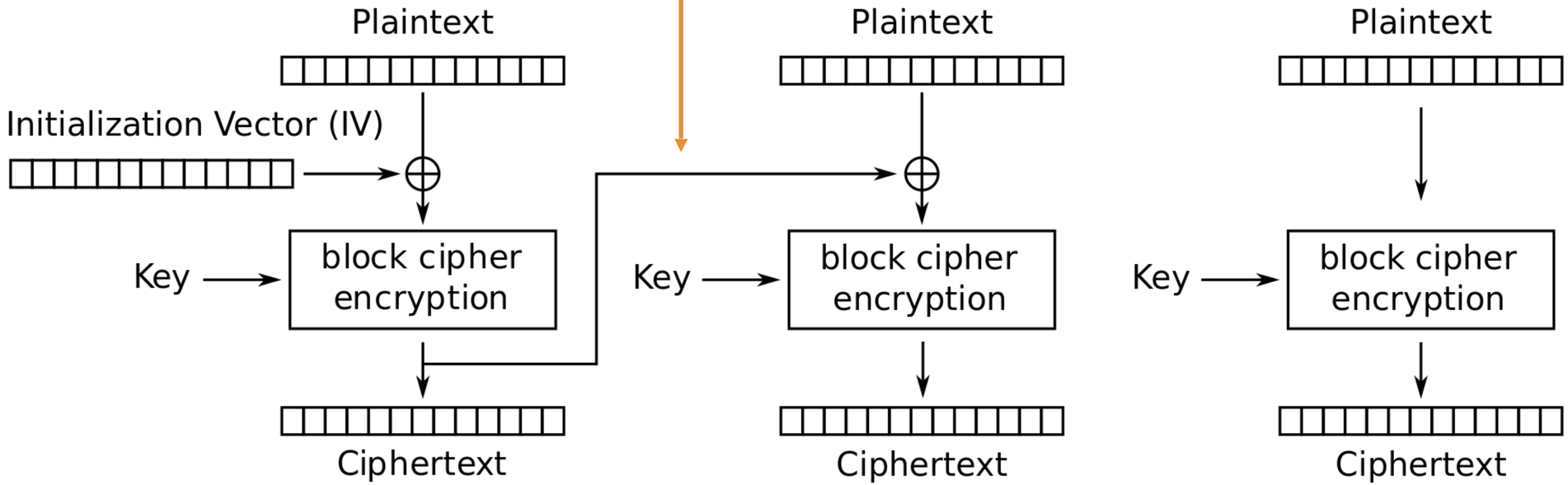
The **Initialization Vector** (**IV**) is different for every encryption. Now the first ciphertext block will be different for every encryption!

Okay, but the other blocks are still deterministic...

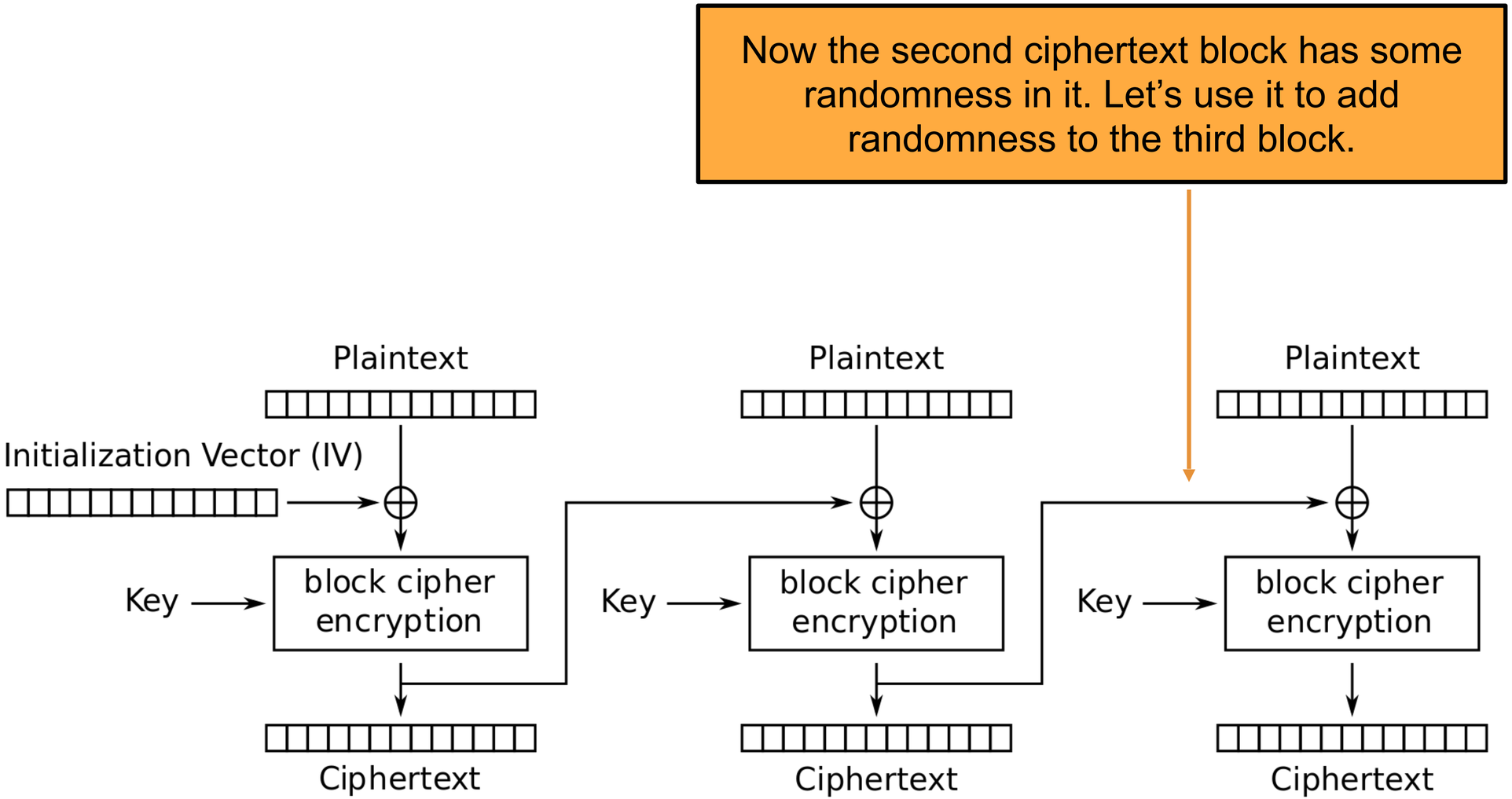


# Scratchpad: Let's design it together

Idea: The first ciphertext block was computed with some randomness. Let's use it to add randomness to the second block.

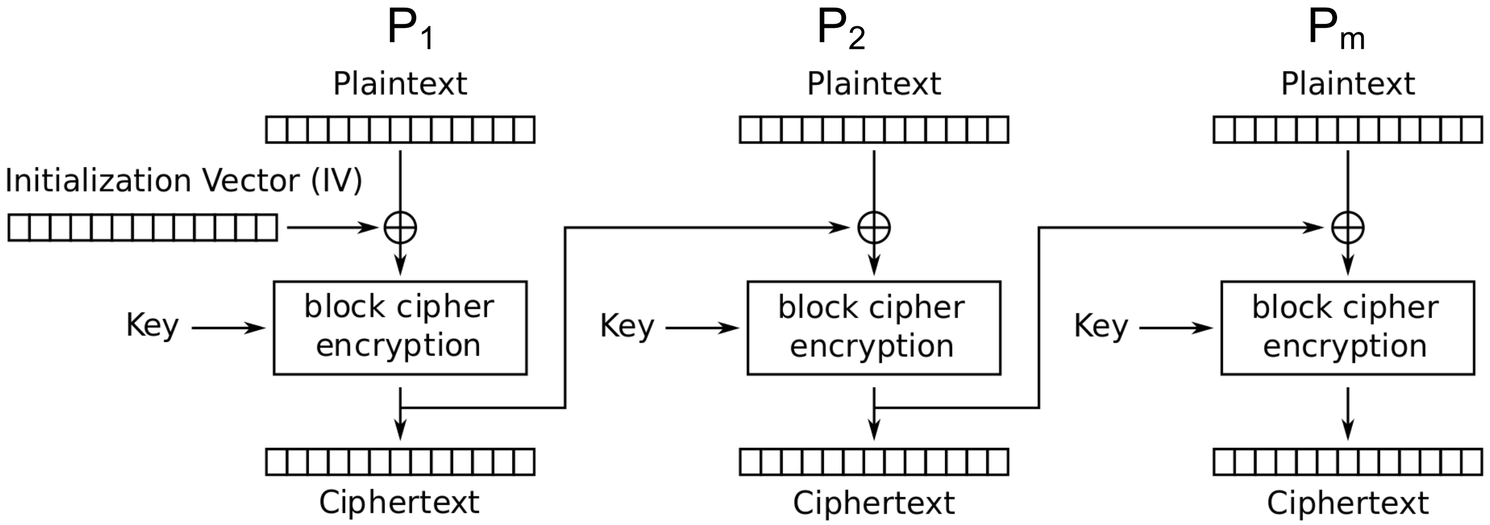


# Scratchpad: Let's design it together



# CBC Mode

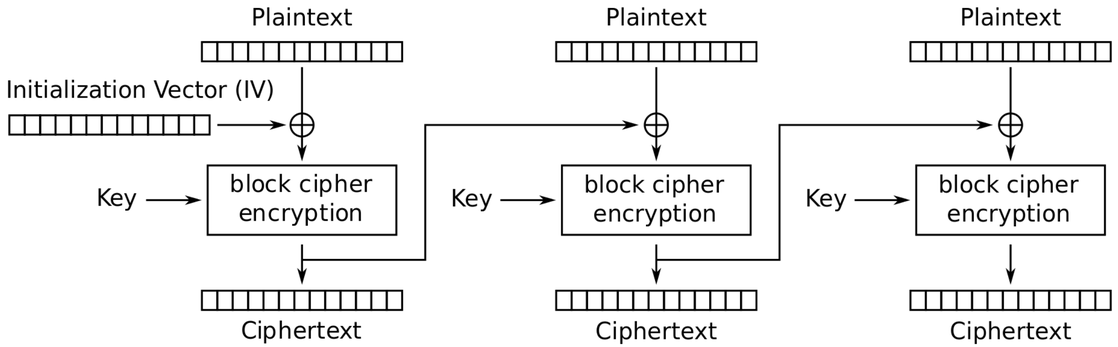
- We've just designed **cipher block chaining (CBC) mode**
- $C_i = E_K(M_i \oplus C_{i-1})$; $C_0 = IV$
- Enc(K, M):
  - Split M in m plaintext blocks $P_1 \ldots P_m$ each of size n
  - Choose a random IV
  - Compute and output (IV, $C_1$, …, $C_m$) as the overall ciphertext
- How do we decrypt?



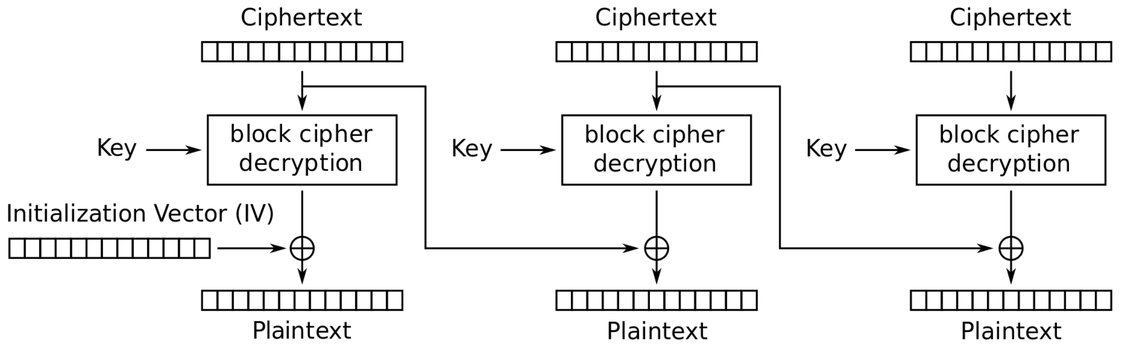

Cipher Block Chaining (CBC) mode encryption

# CBC Mode: Decryption

- How do we decrypt CBC mode?
  - Parse ciphertext as (IV, $C_1$, …, $C_m$)
  - Decrypt each ciphertext and then XOR with IV or previous ciphertext




Cipher Block Chaining (CBC) mode encryption




Cipher Block Chaining (CBC) mode decryption

# CBC Mode: Decryption

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

$$D_K(C_i) = D_K(E_K(M_i \oplus C_{i-1}))$$ Decrypting both sides

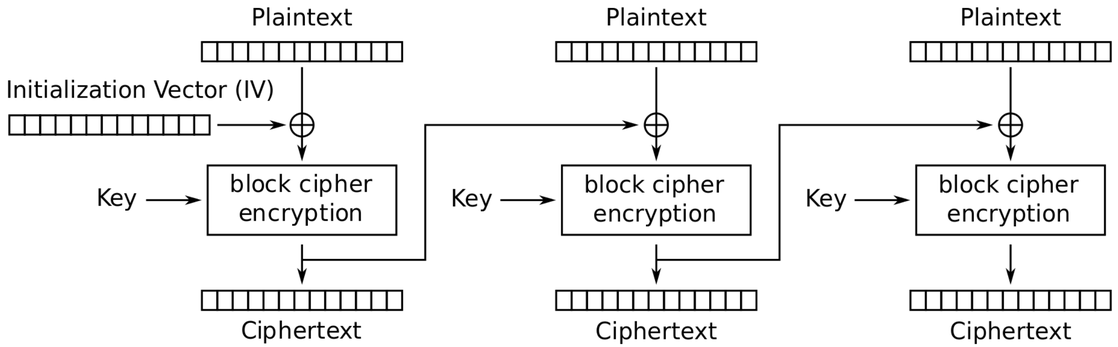
$$D_K(C_i) = M_i \oplus C_{i-1}$$ Decryption and encryption cancel

$$D_K(C_i) \oplus C_{i-1} = M_i \oplus C_{i-1} \oplus C_{i-1}$$ XOR both sides with $C_{i-1}$

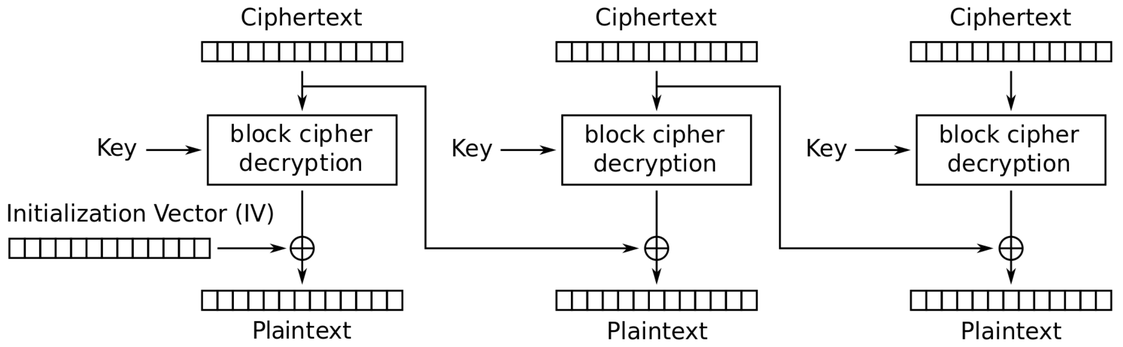
$$D_K(C_i) \oplus C_{i-1} = M_i$$ XOR property

# CBC Mode: Efficiency & Parallelism

- Can encryption be parallelized?
  - No, we have to wait for block i to finish before encrypting block i+1
- Can decryption be parallelized?
  - Yes, decryption only requires ciphertext as input



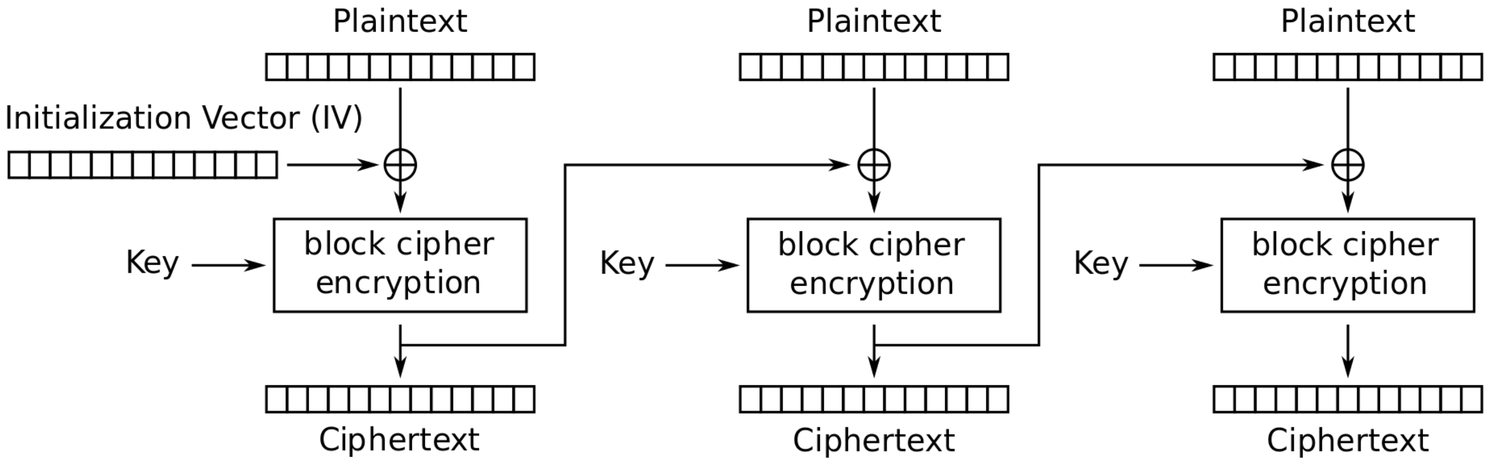

Cipher Block Chaining (CBC) mode encryption




Cipher Block Chaining (CBC) mode decryption

# CBC Mode: Padding

- What if you want to encrypt a message that isn't a multiple of the block size?
  - AES-CBC is only defined if the plaintext length is a multiple of the block size
- Solution: Pad the message until it's a multiple of the block size
  - **Padding**: Adding dummy bytes at the end of the message until it's the proper length




Cipher Block Chaining (CBC) mode encryption

# CBC Mode: Padding

- What padding scheme should we use?
  - Padding with 0's?
    - Doesn't work: What if our message already ends with 0's?
  - Padding with 1's?
    - Same problem
- We need a scheme that can be unpadded without ambiguity
  - One scheme that works: Append a 1, then pad with 0's
    - If plaintext is multiple of n, you still need to pad with an entire block
  - Another scheme: Pad with the number of padding bytes
    - So if you need 1 byte, pad with **01**; if you need 3 bytes, pad with **03 03 03**
    - If you need 0 padding bytes, pad an entire dummy block
    - This is called PKCS #7

# CBC Mode: Padding

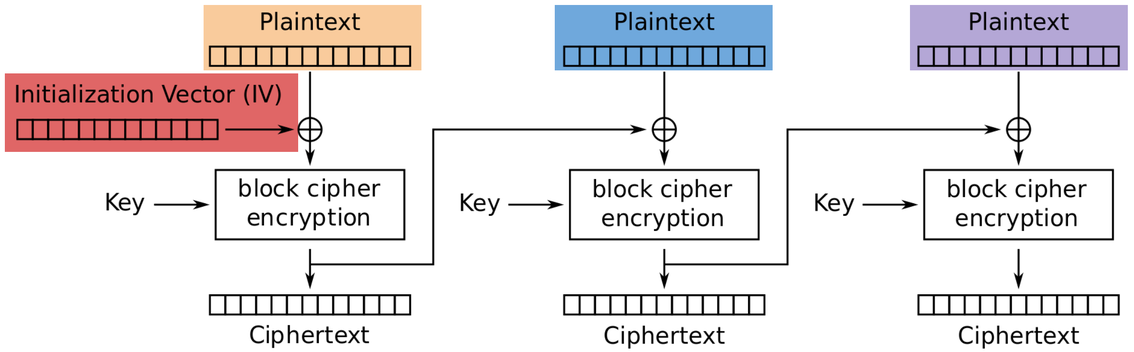
- PKCS #7
  - Given a block size of B bytes (e.g., 8), and a plaintext of M bytes
  - if M is a multiple of B, then add B bytes of B values as the padding
    - When B = 8, and M = 64, then 8 bytes of padding will be appended: **08 08 08 08 08 08 08 08**
  - If M is not a integer multiple of B, then N bytes will be added to make M + N the nearest multiple of B
    - Each padding byte is assigned with a value of N
    - When B = 8, and M = 45: then N = 3, and the padding is **03 03 03**
    - When B = 16, and M = 54, then N = 10, and the padding is **0A 0A 0A 0A 0A 0A 0A 0A 0A 0A**
  - **It only works when B <= 255**
- Other padding schemes
  - ANSI X9.23 Padding:  given N bytes needed, N - 1 bytes as  zero and the last byte as N
  - OneAndZeroes Padding: Ox80 + zero bytes

# CBC Mode: Security

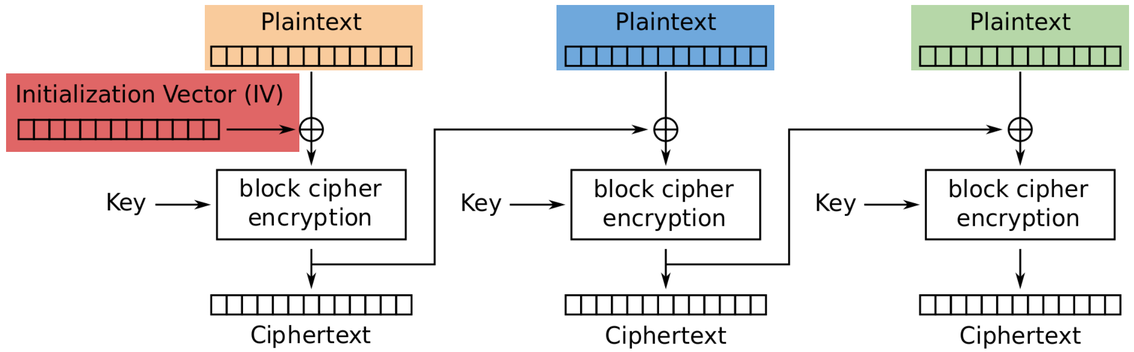
- AES-CBC is IND-CPA secure. With what assumption?
  - The IV must be randomly generated and never reused
- What happens if you reuse the IV?
  - The scheme becomes deterministic: No more IND-CPA security

# CBC Mode: IV Reuse

- Consider two three-block messages: $P_1P_2P_3$ and $P_1P_2P_4$
  - The first two blocks are the same for both messages, but the last block is different
  - What if we encrypt them with the same IV?
- When the IV is reused, CBC mode reveals when two messages start with the same plaintext blocks, up to the first different plaintext block




Cipher Block Chaining (CBC) mode encryption

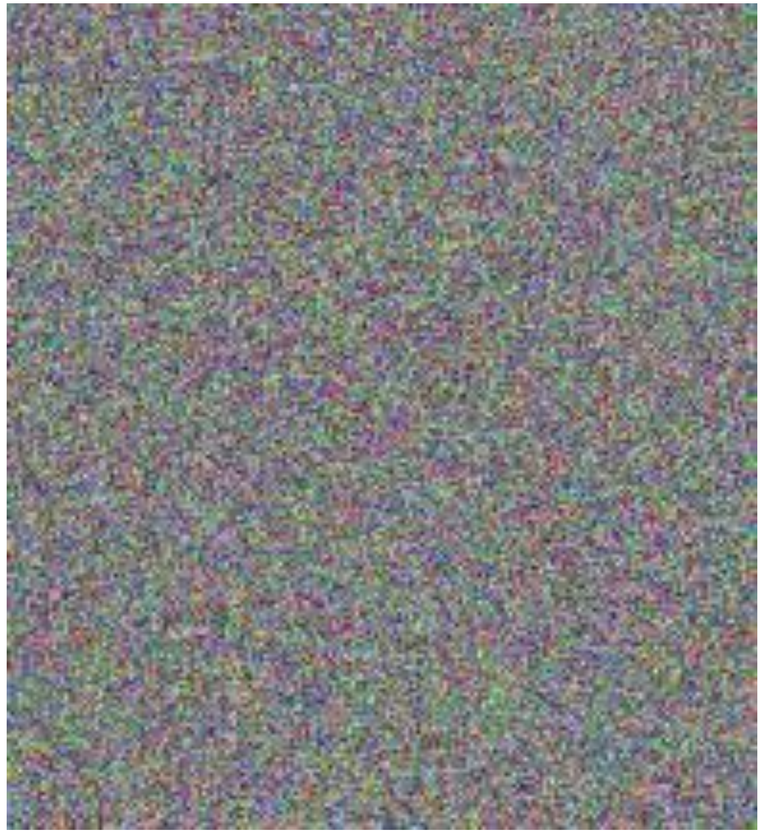



Cipher Block Chaining (CBC) mode encryption

# CBC Mode: Penguin

Original image

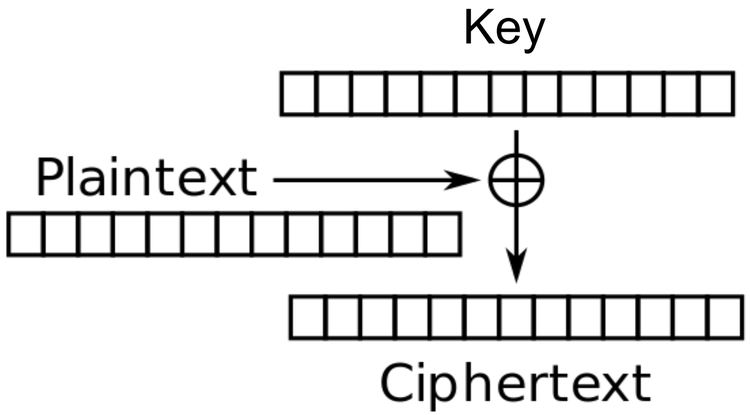
# CBC Mode: Penguin



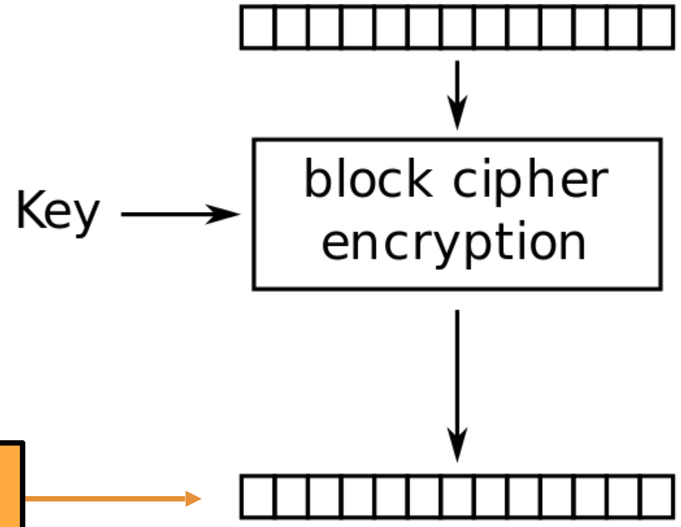
Encrypted with CBC, with random IVs

# CTR Mode Scratchpad: Let's design it together

One-time pads are secure if we never reuse the key.



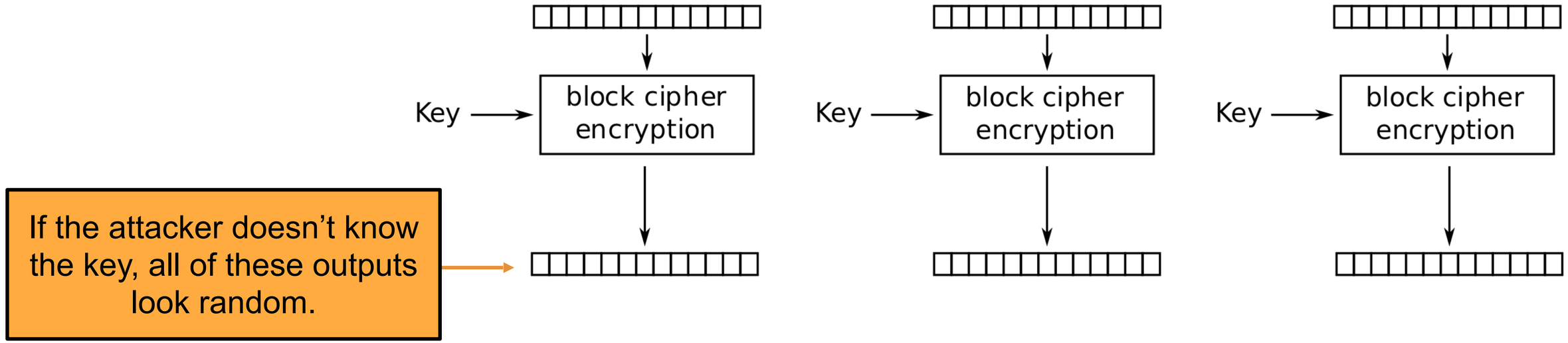
# CTR Mode Scratchpad: Let's design it together



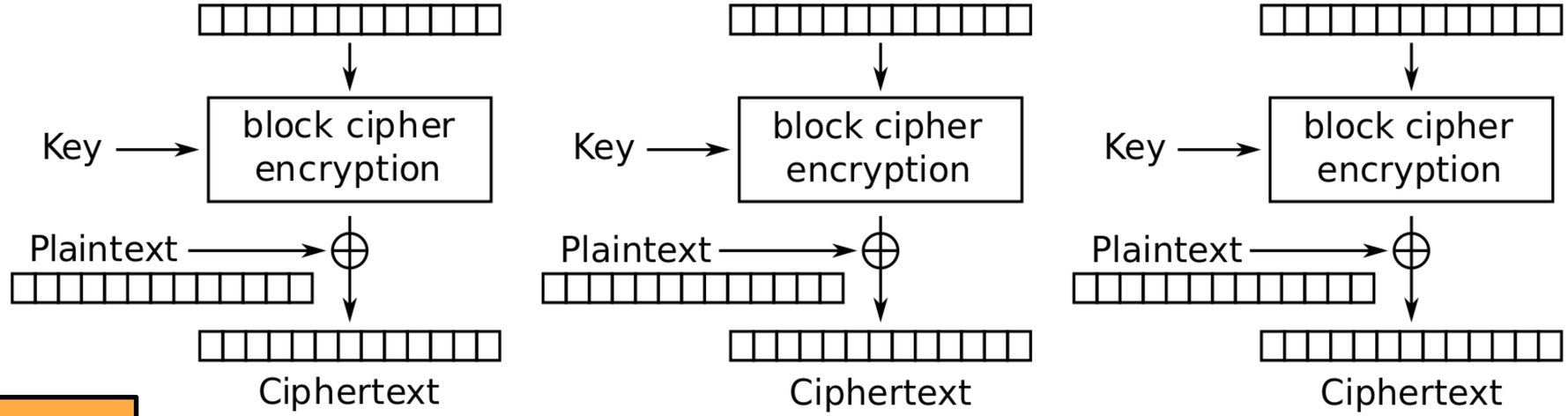

If the attacker doesn't know the key, the block cipher output looks random.

New idea: Can we use block ciphers to simulate a one-time pad?

# CTR Mode Scratchpad: Let's design it together



# CTR Mode Scratchpad: Let's design it together

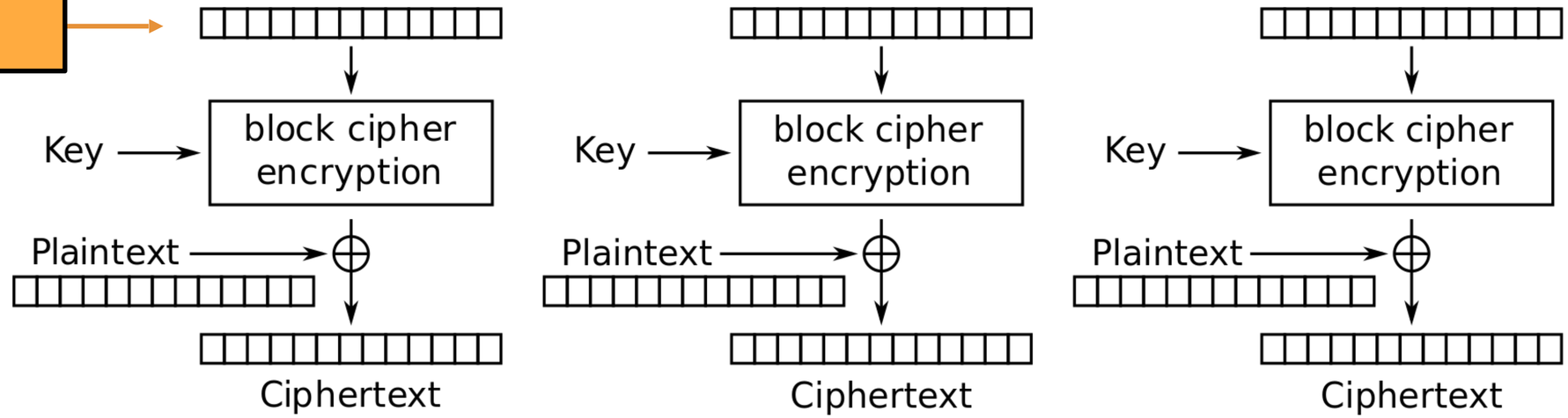



Idea: Use this random-looking output as a one-time pad!

Remember one-time pads: XOR the pad with plaintext to get ciphertext

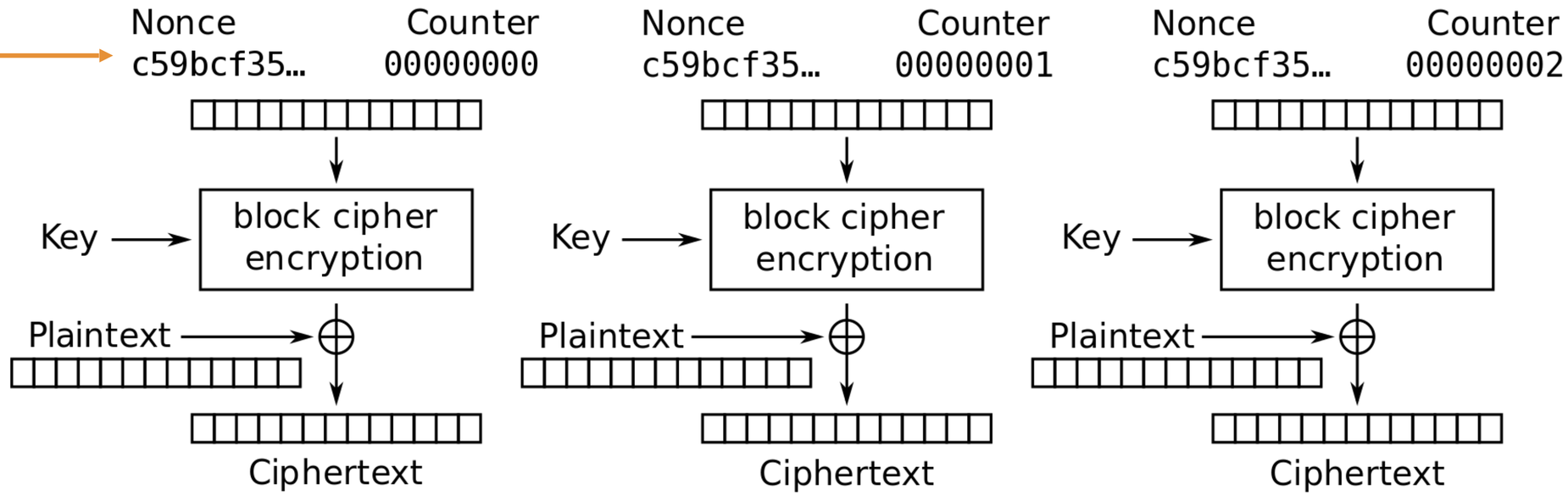
# CTR Mode Scratchpad: Let's design it together

What do we use as input to the block cipher?



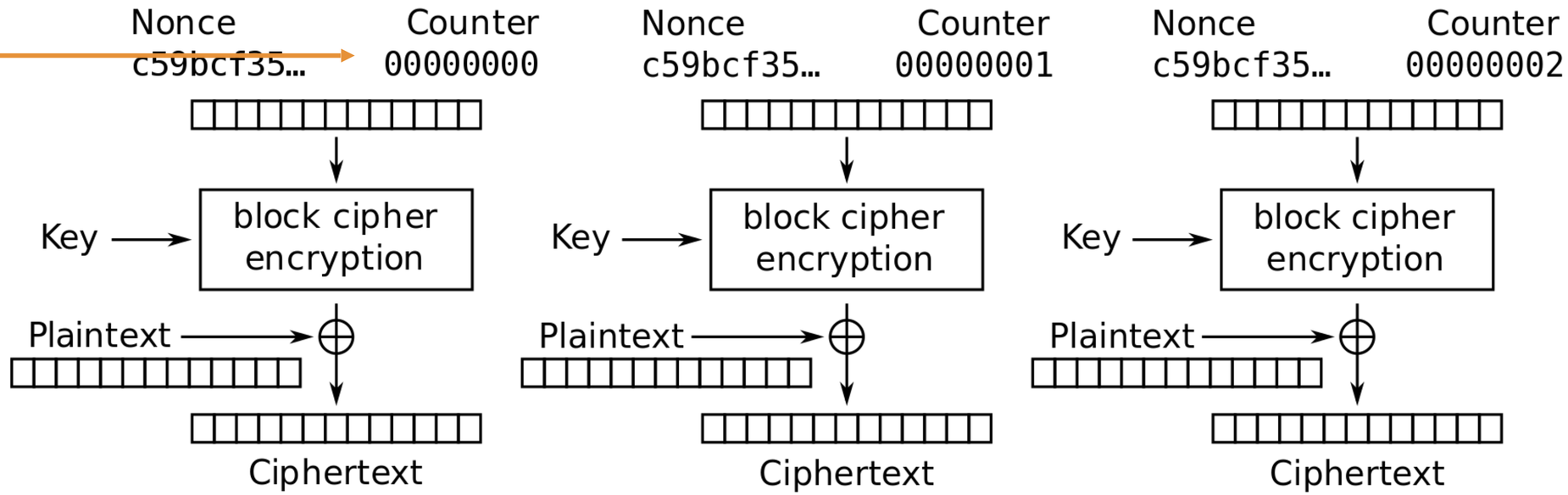
# CTR Mode Scratchpad: Let's design it together

IND-CPA schemes need randomness, so let's put a random **nonce** here!



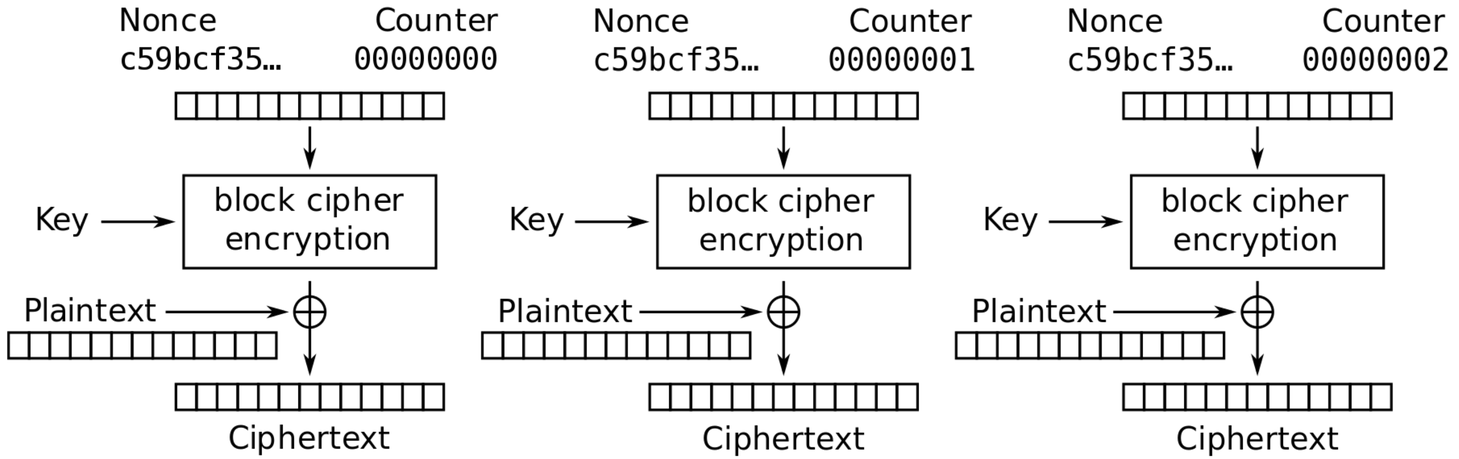
# CTR Mode Scratchpad: Let's design it together

The **counter** increments per block to ensure each block cipher output is different.



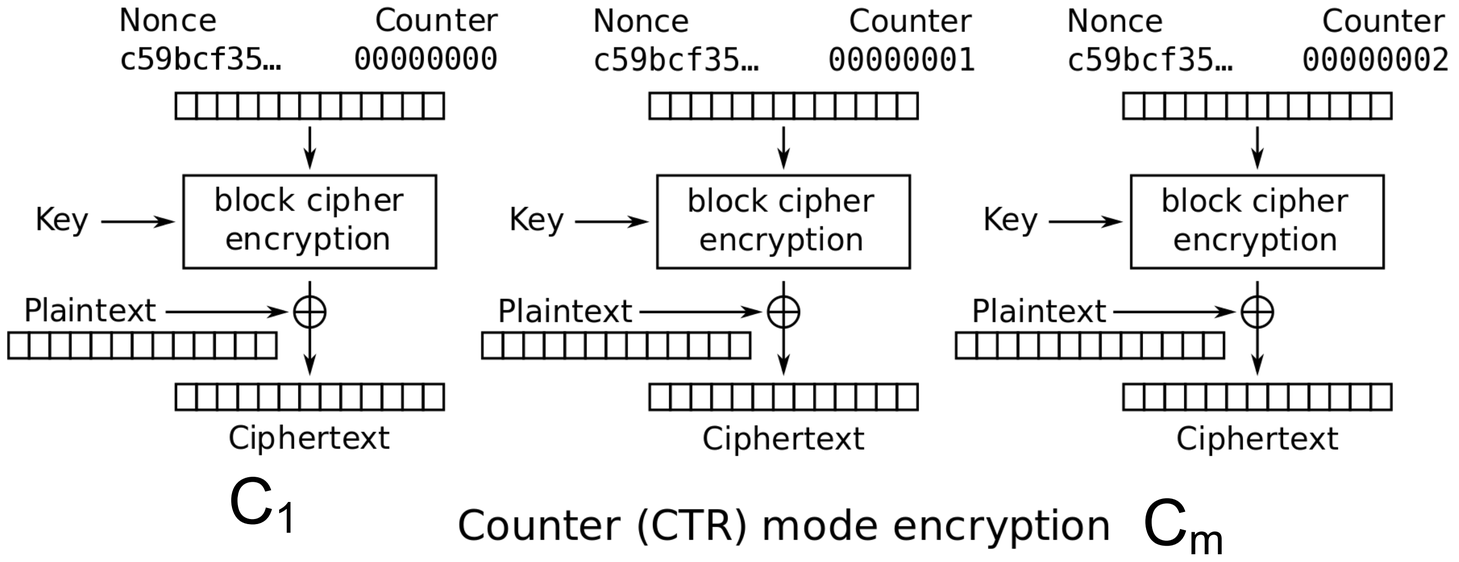
# CTR (Counter) Mode

- Note: the random value is named the nonce here, but the idea is the same as the IV in CBC mode
- Overall ciphertext is (Nonce, $C_1$, …, $C_m$)



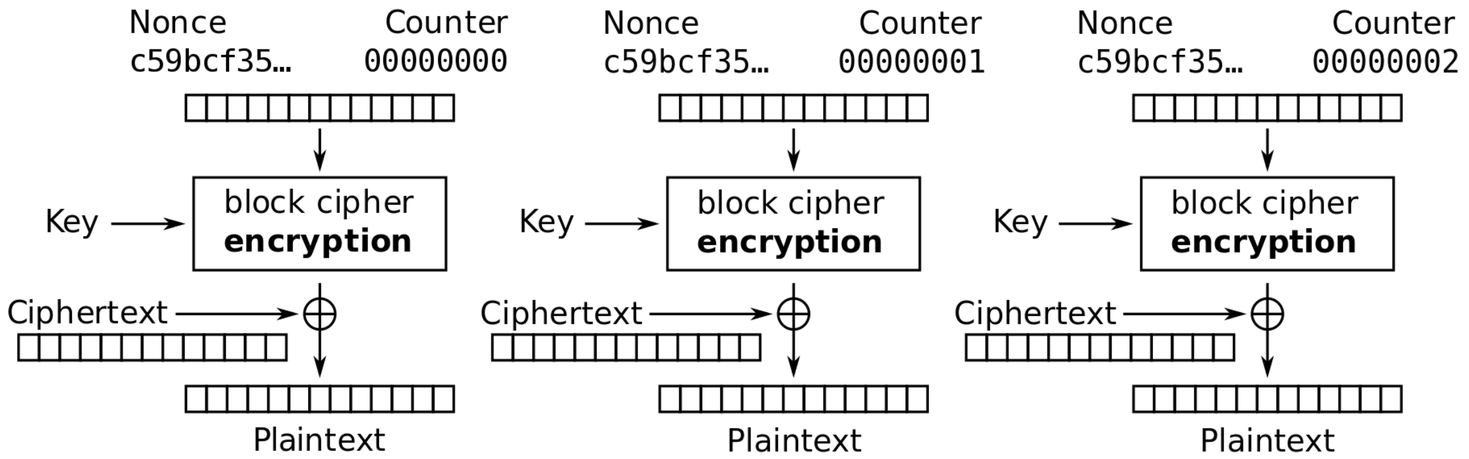
# CTR Mode

- Enc(K, M):
  - Split M in plaintext blocks $P_1...P_m$ (each of block size n)
  - Choose random nonce
  - Compute and output (Nonce, $C_1$, …, $C_m$)
- How do you decrypt?



# CTR Mode: Decryption

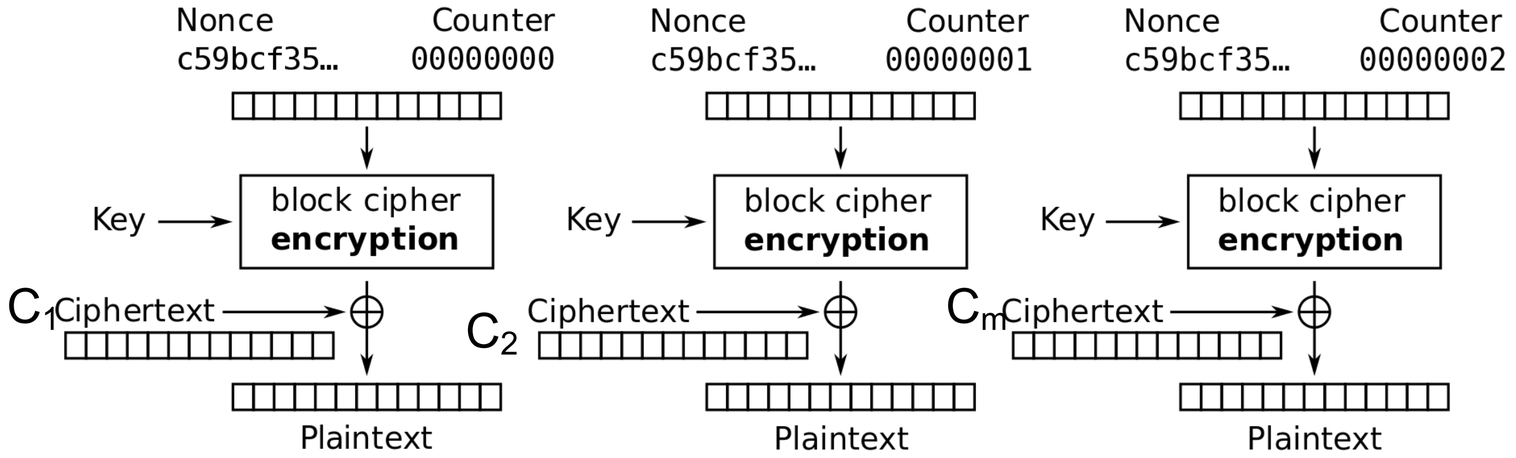
- Recall one-time pad: XOR with ciphertext to get plaintext
- Note: we are only using block cipher encryption, not decryption




Counter (CTR) mode decryption

# CTR Mode: Decryption

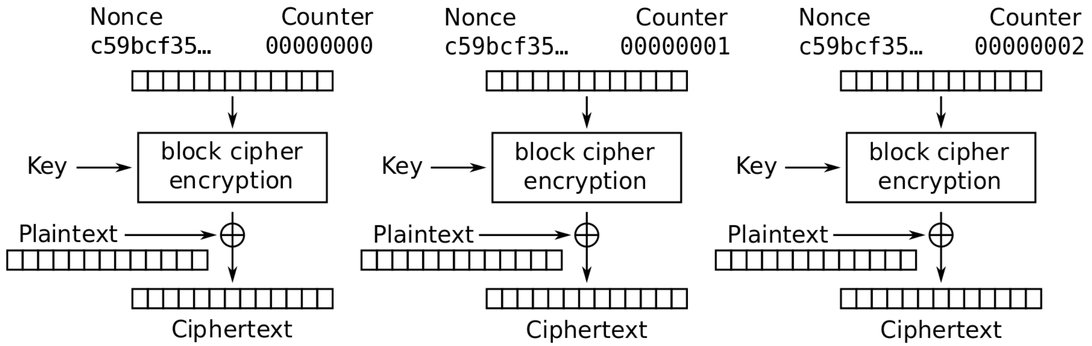
- Dec(K, C):
  - Parse C into (nonce, $C_1$, …, $C_m$)
  - Compute $P_i$ by XORing Ci with output of $E_k$ on nonce and counter
  - Concatenate resulting plaintexts and output M = $P_1$ … $P_m$



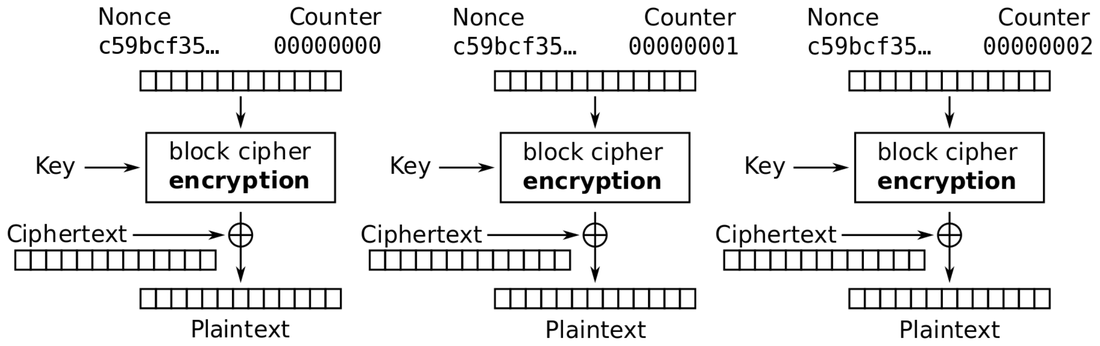

Counter (CTR) mode decryption

# CTR Mode: Efficiency

- Can encryption be parallelized?
  - Yes
- Can decryption be parallelized?
  - Yes



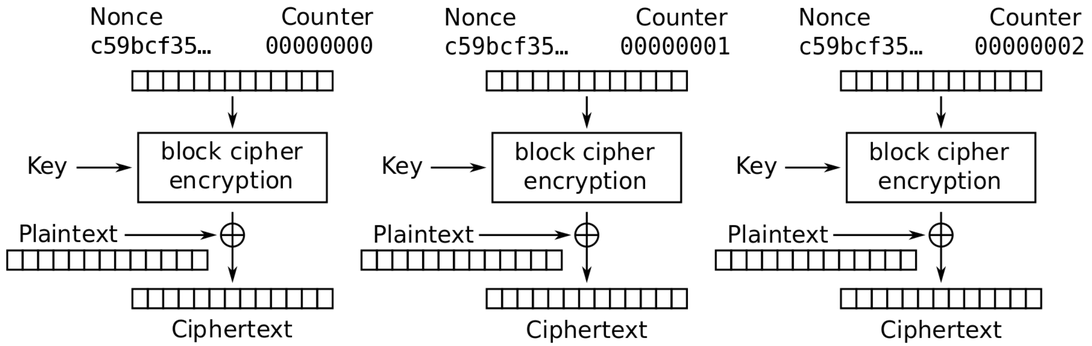

Counter (CTR) mode encryption



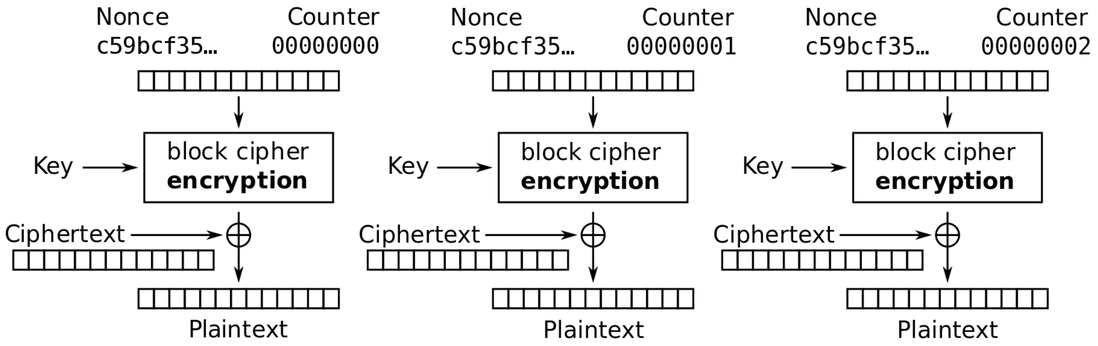

Counter (CTR) mode decryption

# CTR Mode: Padding

- Do we need to pad messages?
  - No! We can just cut off the parts of the XOR that are longer than the message.




Counter (CTR) mode encryption




Counter (CTR) mode decryption

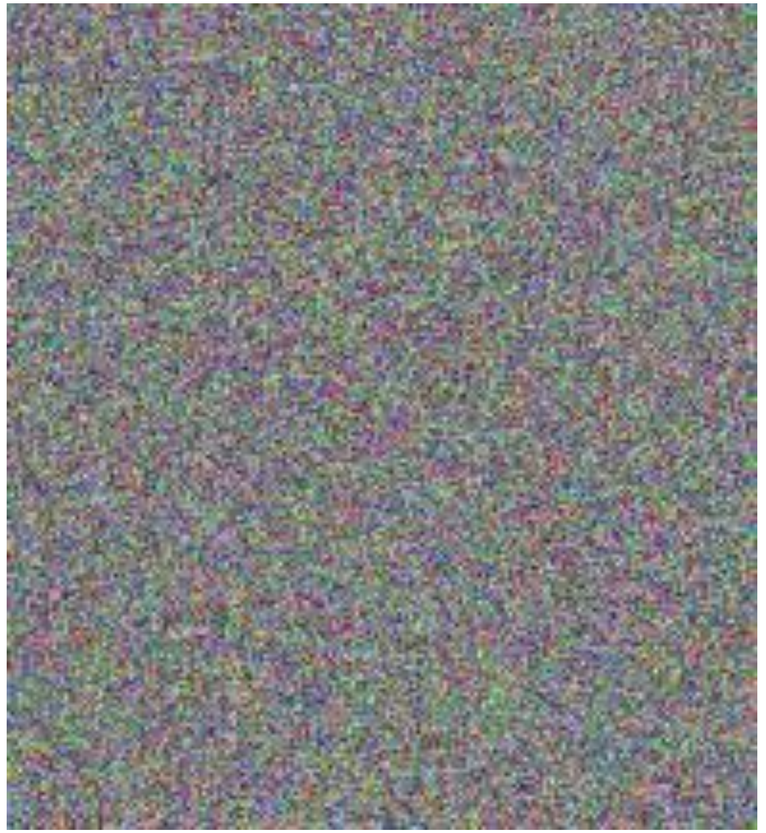
# CTR Mode: Security

- AES-CTR is IND-CPA secure. With what assumption?
- The nonce must be randomly generated and never reused
  - And in general less than $2^{n/2}$ blocks are encrypted
- What happens if you reuse the nonce?
- Equivalent to reusing a key in a one-time pad
  - Recall: Key reuse in a one-time pad is catastrophic: usually leaks enough information for an attacker to deduce the entire plaintext

# CTR Mode: Penguin

Original image

# CTR Mode: Penguin



Encrypted with CTR, with random nonces

# IVs and Nonces

- **Initialization vector** (**IV**): A random, but public, one-use value to introduce randomness into the algorithm
  - For CTR mode, we say that you use a **nonce** (number used once), since the value has to be unique, not necessarily random.
  - In this class, we use IV and nonce interchangeably
- **Never reuse IVs**
  - In some algorithms, IV/nonce reuse leaks limited information (e.g. CBC)
  - In some algorithms, IV/nonce reuse leads to catastrophic failure (e.g. CTR)

# IVs and Nonces

- Thinking about the consequences of IV/nonce reuse is hard
- Solution: Randomly generate a new IV/nonce for every encryption
  - If the nonce is 128 bits or longer, the probability of generating the same IV/nonce twice is astronomically small (basically 0)
  - Now you don’t ever have to think about IV/nonce reuse attacks!

# Comparing Modes of Operation

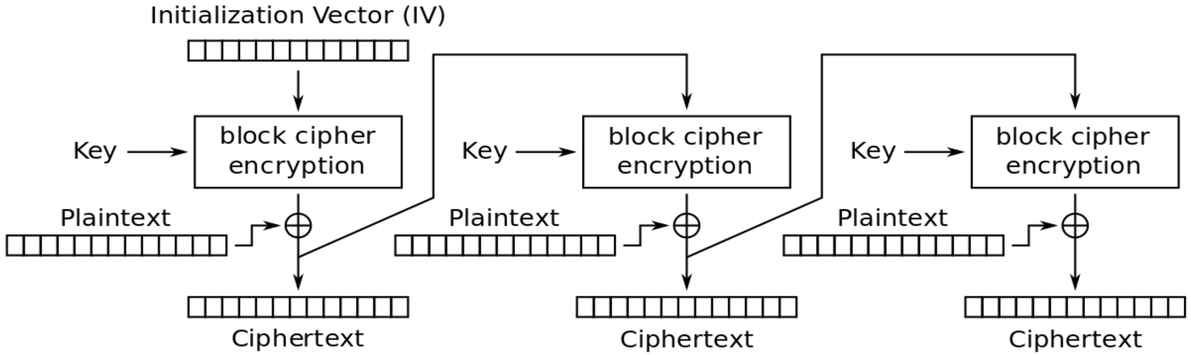
- If you need high performance, which mode is better?
  - CTR mode, because you can parallelize both encryption and decryption
- If you're paranoid about security, which mode is better?
  - CBC mode is better
- Theoretically, CBC and CTR mode are equally secure if used properly
  - However, if used improperly (IV/nonce reuse), CBC only leaks partial information, and CTR fails catastrophically
    - Consider human factors: Systems should be as secure as possible even when implemented *incorrectly*
  - IV failures on CTR mode have resulted in multiple real-world security incidents!

# Other Modes of Operation

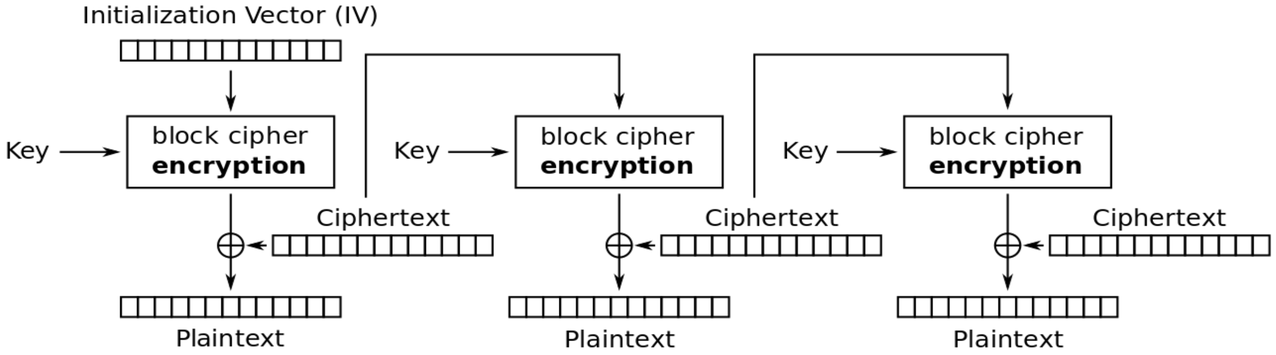
- Other modes exist besides CBC and CTR
- Trade-offs:
  - Do we need to pad messages?
  - How robust is the scheme if we use it incorrectly?
  - Can we parallelize encryption/decryption?

# CFB Mode

- Also IND-CPA
- Try to analyze the trade-offs yourself:
  - Do we need to pad messages?
  - How robust is the scheme if we use it incorrectly?
  - Can we parallelize encryption/decryption?




Cipher Feedback (CFB) mode encryption




Cipher Feedback (CFB) mode decryption

# CFB Mode

- Try to analyze the trade-offs yourself:
  - Do we need to pad messages?
    - No
  - How robust is the scheme if we use it incorrectly?
    - Similar effects as CBC mode
  - Can we parallelize encryption/decryption?
    - Only decryption is parallelizable

# Lack of Integrity and Authenticity

- Block ciphers are designed for *confidentiality* (IND-CPA)
- If an attacker tampers with the ciphertext, we are not guaranteed to detect it
- Remember Mallory: An *active* manipulator who wants to tamper with the message

# Lack of Integrity and Authenticity

- Consider CTR mode
- What if Mallory tampers with the ciphertext using XOR?

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | a | y | | M | a | l | | $ | 1 | 0 | 0 |
| *M* | 0x50 | 0x61 | 0x79 | 0x20 | 0x4d | 0x61 | 0x6c | 0x20 | 0x24 | 0x31 | 0x30 | 0x30 |
| | ⊕ | | | | | | | | | | | |
| $E_K(i)$ | 0x8a | 0xe3 | 0x5e | 0xcf | 0x3b | 0x40 | 0x46 | 0x57 | 0xb8 | 0x69 | 0xd2 | 0x96 |
| | = | | | | | | | | | | | |
| *C* | 0xda | 0x82 | 0x27 | 0xef | 0x76 | 0x21 | 0x2a | 0x77 | 0x9c | 0x58 | 0xe2 | 0xa6 |

# Lack of Integrity and Authenticity

- Suppose Mallory knows the message *M*
- How can Mallory change the *M* to say **Pay Mal $900**?

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | a | y | | M | a | l | | $ | 1 | 0 | 0 |
| *M* | 0x50 | 0x61 | 0x79 | 0x20 | 0x4d | 0x61 | 0x6c | 0x20 | 0x24 | **0x31** | 0x30 | 0x30 |
| | ⊕ | | | | | | | | | | | |
| $E_K(i)$ | 0x8a | 0xe3 | 0x5e | 0xcf | 0x3b | 0x40 | 0x46 | 0x57 | 0xb8 | **0x69** | 0xd2 | 0x96 |
| | = | | | | | | | | | | | |
| *C* | 0xda | 0x82 | 0x27 | 0xef | 0x76 | 0x21 | 0x2a | 0x77 | 0x9c | **0x58** | 0xe2 | 0xa6 |

# Lack of Integrity and Authenticity

| | | |
|---|---|---|
| $C_i = M_i \oplus \text{Pad}_i$ | **0x58** = **0x31** $\oplus$ $\text{Pad}_i$ | Definition of CTR |
| $\text{Pad}_i = M_i \oplus C_i$ | $\text{Pad}_i$ = **0x58** $\oplus$ **0x31**<br>= **0x69** | Solve for the *i*th byte of the pad |
| $C'_i = M'_i \oplus \text{Pad}_i$ | $C'_i$ = **0x39** $\oplus$ **0x69**<br>= **0x50** | Compute the changed *i*th byte |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *C* | 0xda | 0x82 | 0x27 | 0xef | 0x76 | 0x21 | 0x2a | 0x77 | 0x9c | **0x58** | 0xe2 | 0xa6 |
| *C'* | 0xda | 0x82 | 0x27 | 0xef | 0x76 | 0x21 | 0x2a | 0x77 | 0x9c | **0x50** | 0xe2 | 0xa6 |

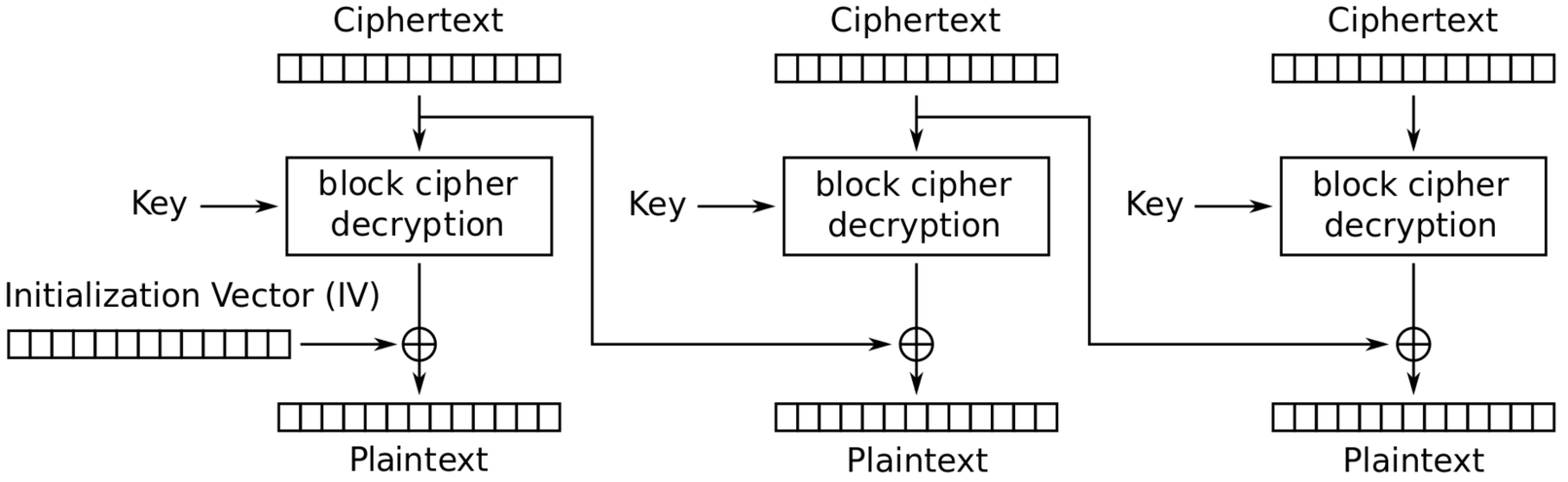
# Lack of Integrity and Authenticity

- What happens when we decrypt *C'*?
  - The message looks like “Pay Mal $900” now!
  - Note: Mallory didn’t have to know the key; no integrity or authenticity for CTR mode!

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *C'* | 0xda | 0x82 | 0x27 | 0xef | 0x76 | 0x21 | 0x2a | 0x77 | 0x9c | **0x50** | 0xe2 | 0xa6 |
| | ⊕ | | | | | | | | | | | |
| $E_K(i)$ | 0x8a | 0xe3 | 0x5e | 0xcf | 0x3b | 0x40 | 0x46 | 0x57 | 0xb8 | **0x69** | 0xd2 | 0x96 |
| | = | | | | | | | | | | | |
| *P'* | 0x50 | 0x61 | 0x79 | 0x20 | 0x4d | 0x61 | 0x6c | 0x20 | 0x24 | **0x39** | 0x30 | 0x30 |
| | P | a | y | | M | a | l | | $ | **9** | 0 | 0 |

# Lack of Integrity and Authenticity

- What about CBC?
  - Altering a bit of the ciphertext causes some blocks to become random gibberish
  - However, Bob cannot prove that Alice did not send random gibberish, so it still does *not* provide integrity or authenticity

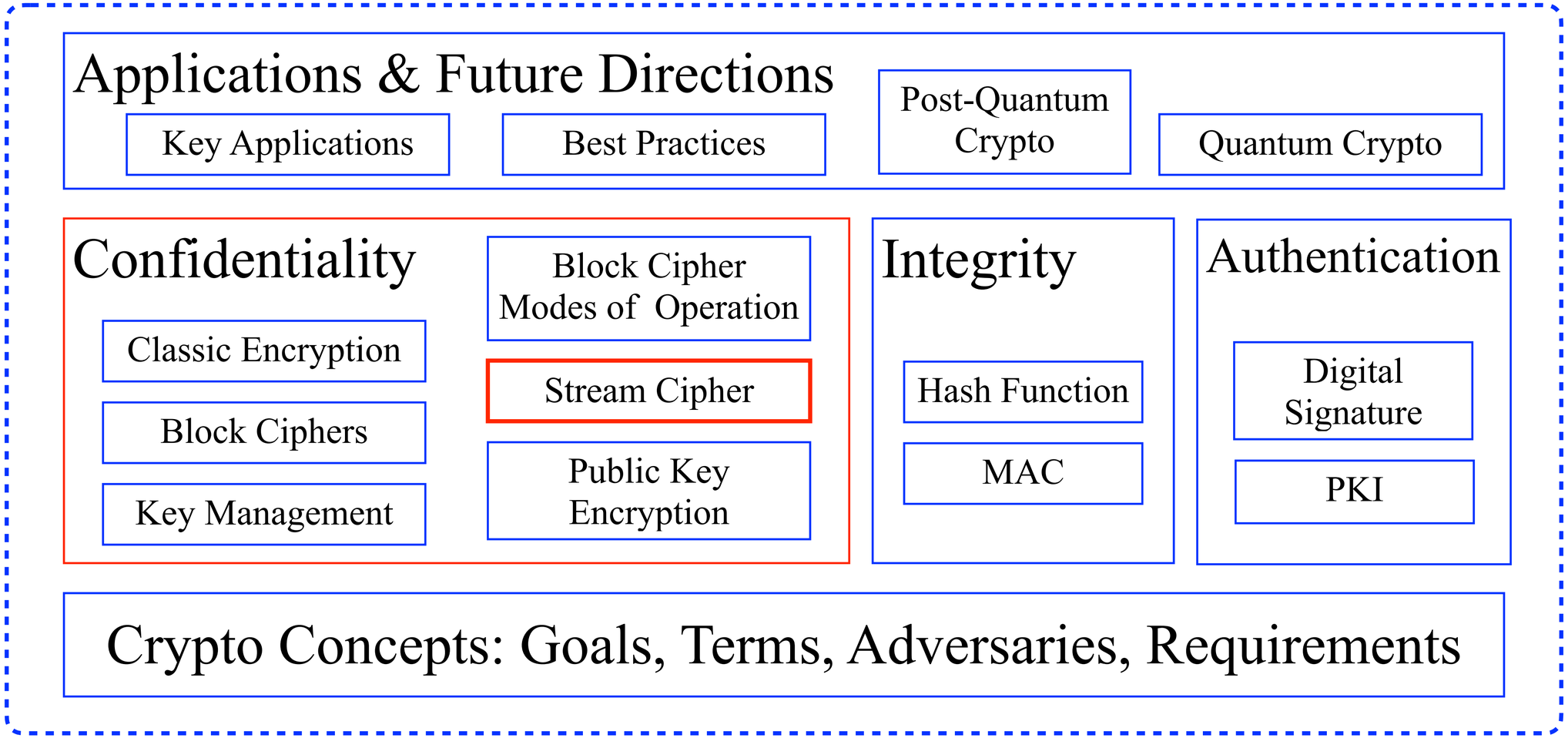



Cipher Block Chaining (CBC) mode decryption

# Block Cipher Modes of Operation: Summary

- ECB mode: Deterministic, so not IND-CPA secure
- CBC mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption is not parallelizable
  - Decryption is parallelizable
  - Must pad plaintext to a multiple of the block size
  - IV reuse leads to leaking the existence of identical blocks at the start of the message
- CTR mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption and decryption are parallelizable
  - Plaintext does not need to be padded
  - Nonce reuse leads to losing all security
- Lack of integrity and authenticity

# The Roadmap of Cryptography

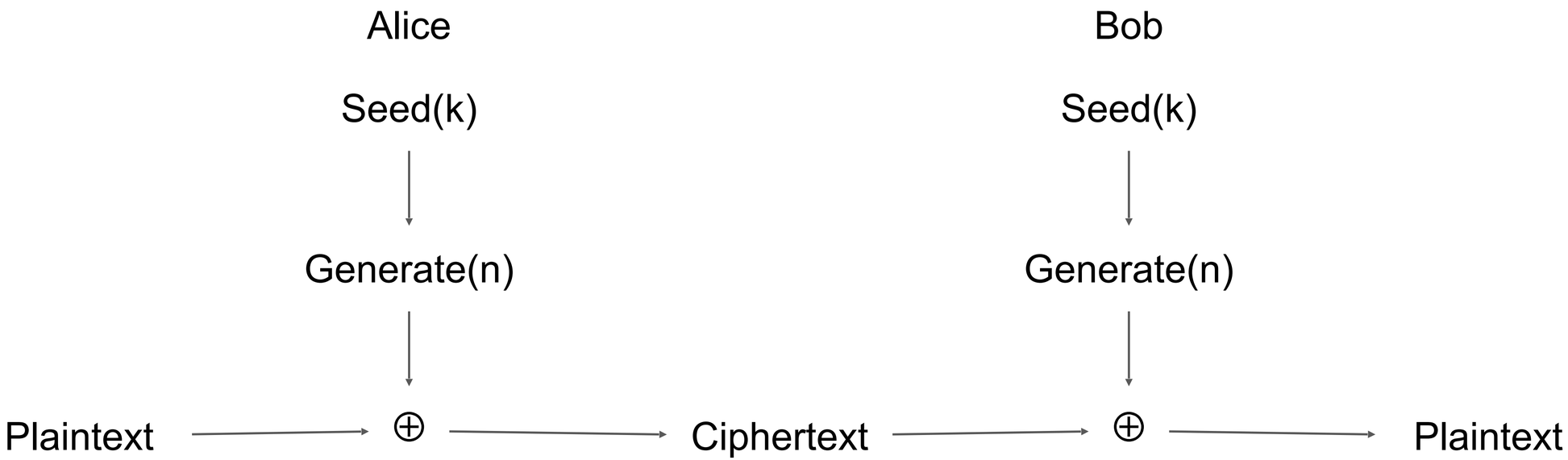


# Stream Ciphers

- **Stream cipher**: A symmetric encryption algorithm that uses pseudorandom bits as the key to a one-time pad
- Idea
  - A secure PRNG (Pseudorandom Number Generators) produces output that looks indistinguishable from random
  - An attacker who can’t see the internal PRNG state can’t learn any output
  - What if we used PRNG output as the key to a one-time pad?

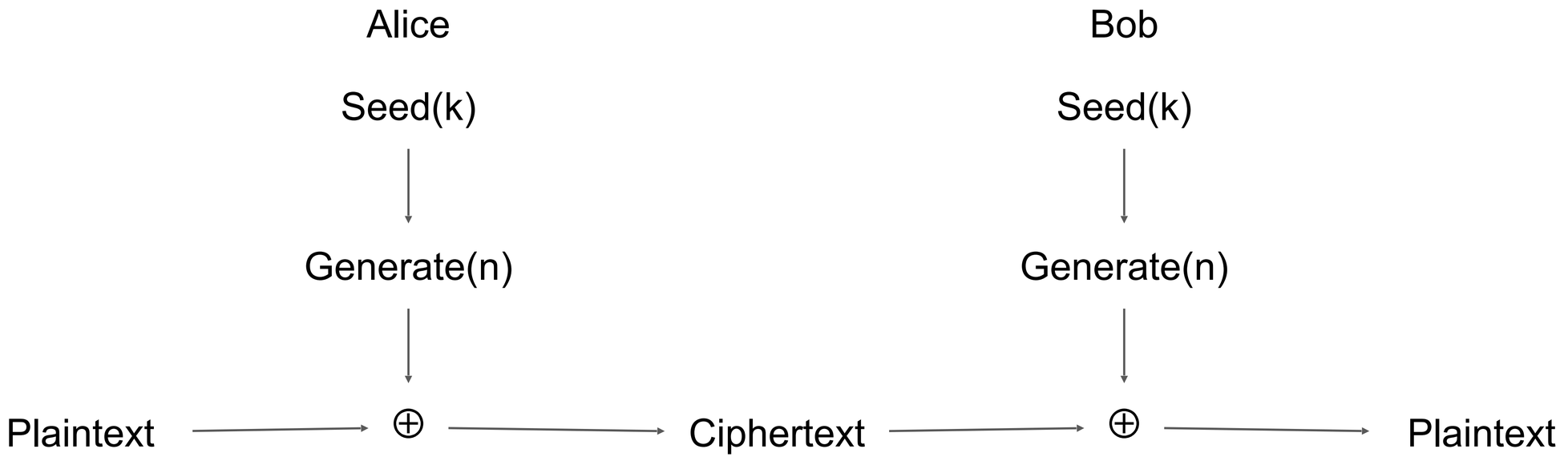
# Stream Ciphers

- Protocol: Alice and Bob both seed a secure PRNG with their symmetric secret key, and then use the output as the key for a one-time pad



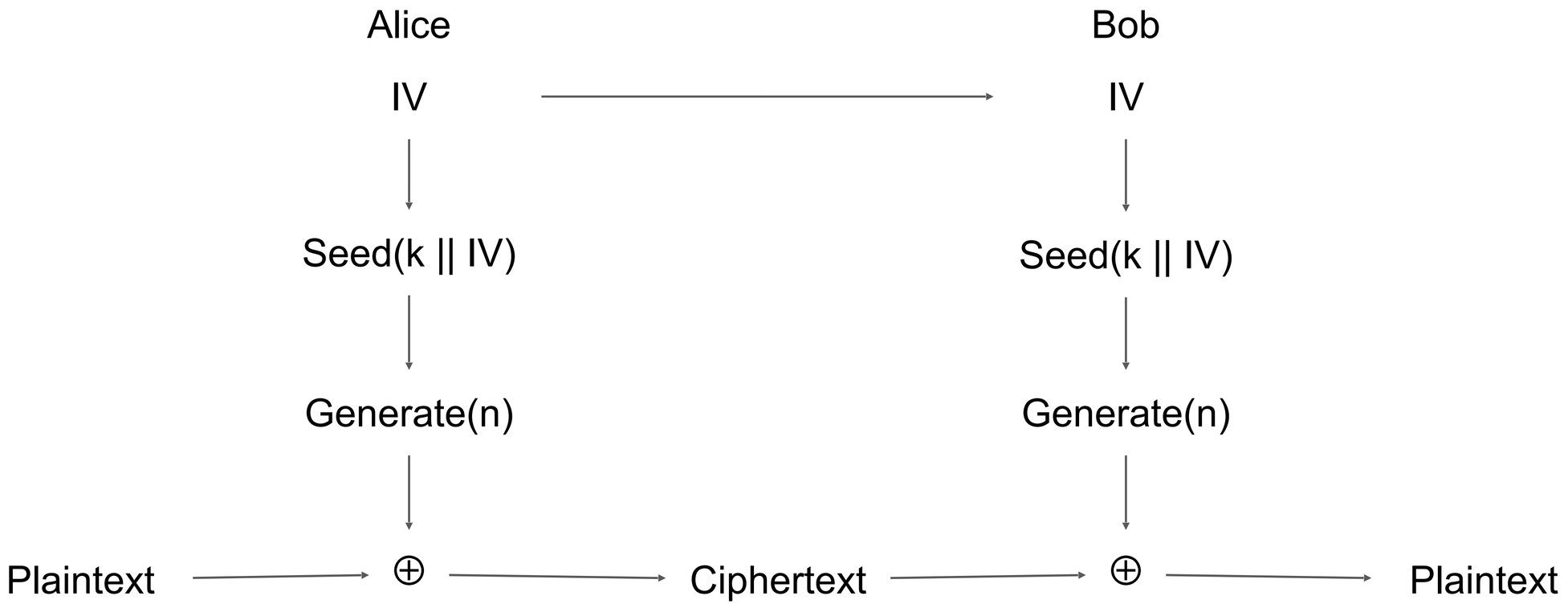
# Stream Ciphers: Encrypting Multiple Messages

- Recall: One-time pads are insecure when the key is reused. How do we encrypt multiple messages without key reuse?



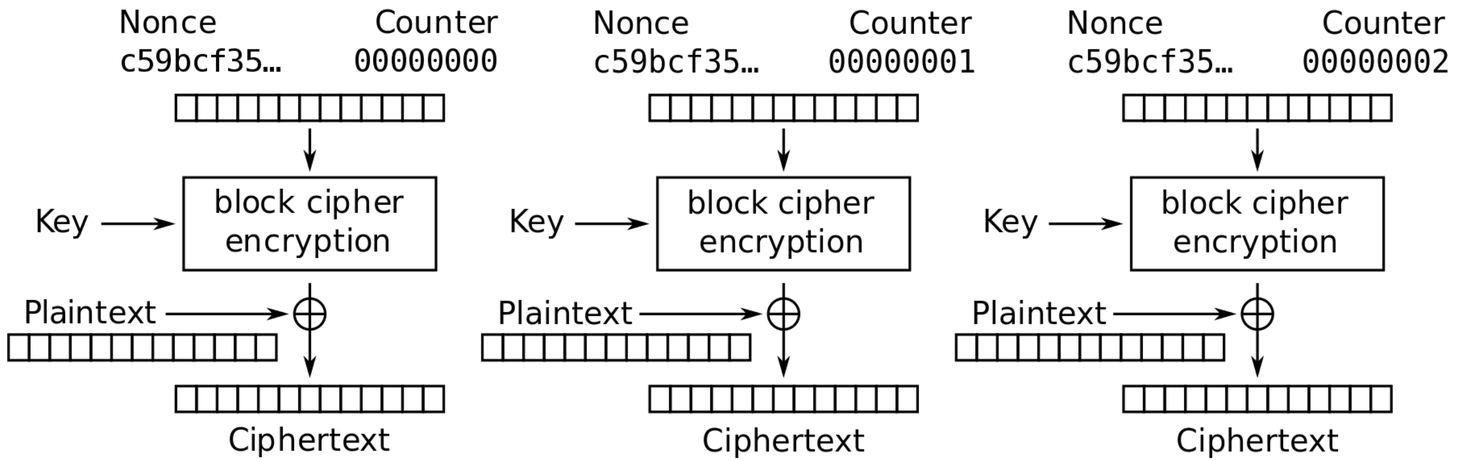
# Stream Ciphers: Encrypting Multiple Messages

- Solution: For each message, seed the PRNG with the key and a random IV, concatenated. Send the IV with the ciphertext



# Stream Ciphers: AES-CTR

- If you squint carefully, AES-CTR is a type of stream cipher
- Output of the block ciphers is pseudorandom and used as a one-time pad




Counter (CTR) mode encryption

# Stream Ciphers: Security

- Stream ciphers are IND-CPA secure, assuming the pseudorandom output is secure
- In some stream ciphers, security is compromised if too much plaintext is encrypted
  - Example: In AES-CTR, if you encrypt so many blocks that the counter wraps around, you'll start reusing keys
  - In practice, if the key is $n$ bits long, usually stop after $2^{n/2}$ bits of output
  - Example: In AES-CTR with 128-bit counters, stop after $2^{64}$ blocks of output

# Stream Ciphers: Encryption Efficiency

- Stream ciphers can continually process new elements as they arrive
  - Only need to maintain internal state of the PRNG
  - Keep generating more PRNG output as more input arrives
- Compare to block ciphers: Need modes of operations to handle longer messages, and modes like AES-CBC need padding to function, so doesn’t function well on streams

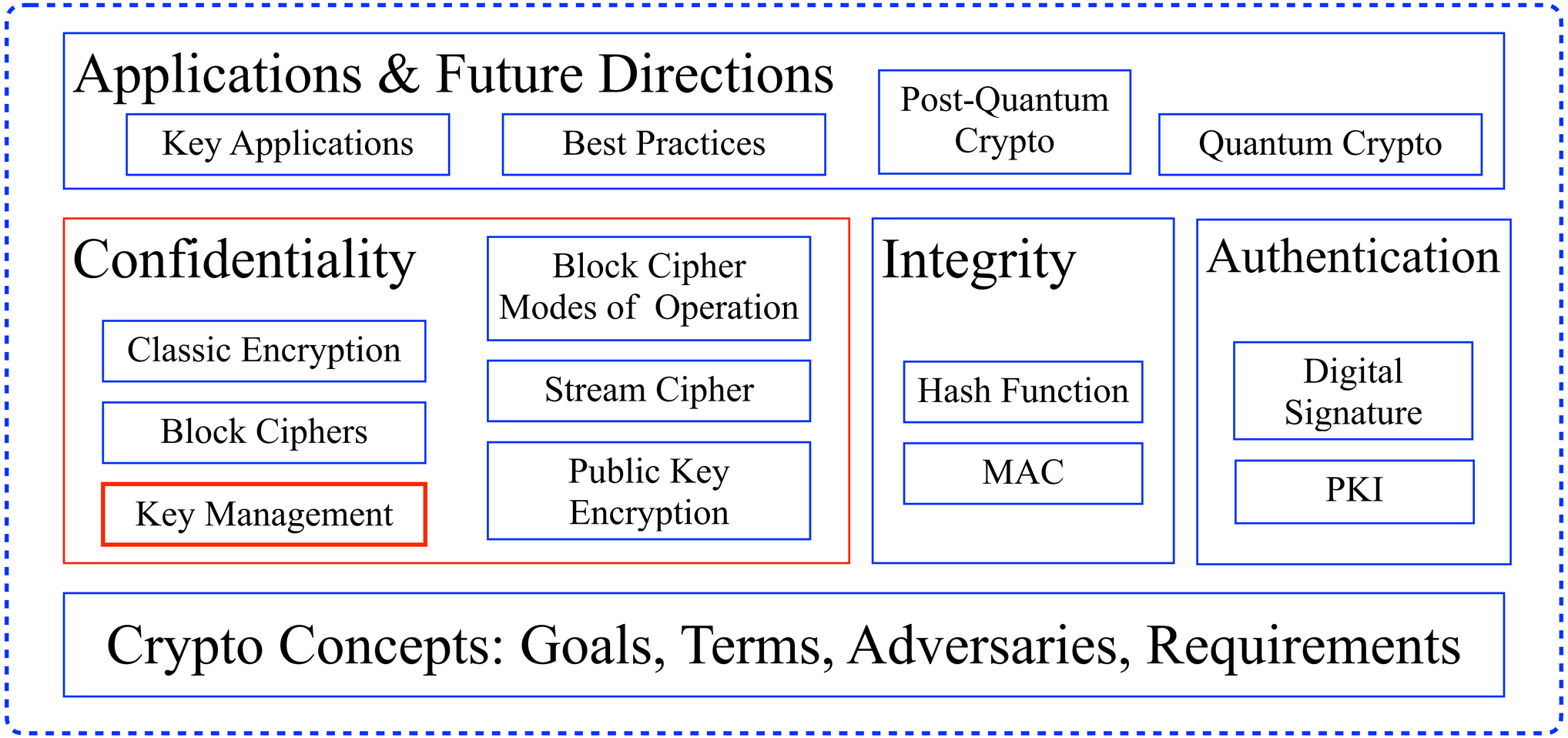
# Stream Ciphers: Decryption Efficiency

- Suppose you received a 1 GB ciphertext (encryption of a 1 GB message) and you only wanted to decrypt the last 128 bytes
- Benefit of some stream ciphers: You can decrypt one part of the ciphertext without decrypting the entire ciphertext
  - Example: In AES-CTR, to decrypt only block *i*, compute $E_K$(nonce || *i*) and XOR with the *i*th block of ciphertext
  - Example: ChaCha20 (another stream cipher) lets you decrypt arbitrary parts of ciphertext

# Next: Diffie-Hellman Key Exchange

- When discussing symmetric-key schemes, we assumed Alice and Bob managed to share a secret key. How can Alice and Bob share a symmetric key over an insecure channel?

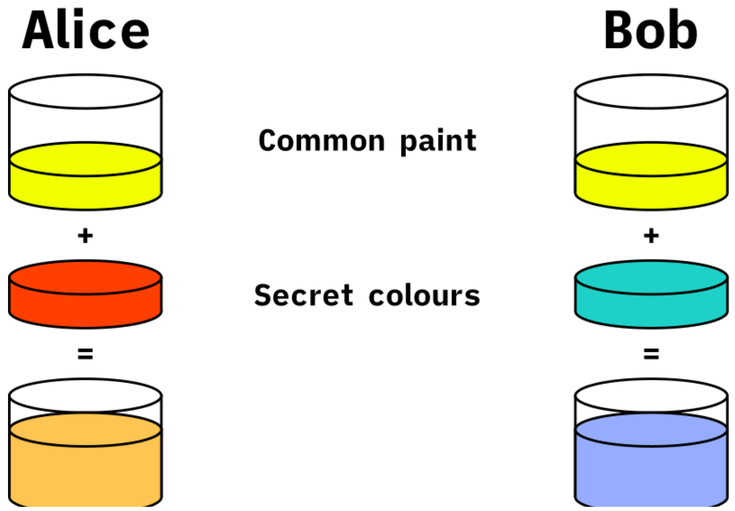
# The Roadmap of Cryptography



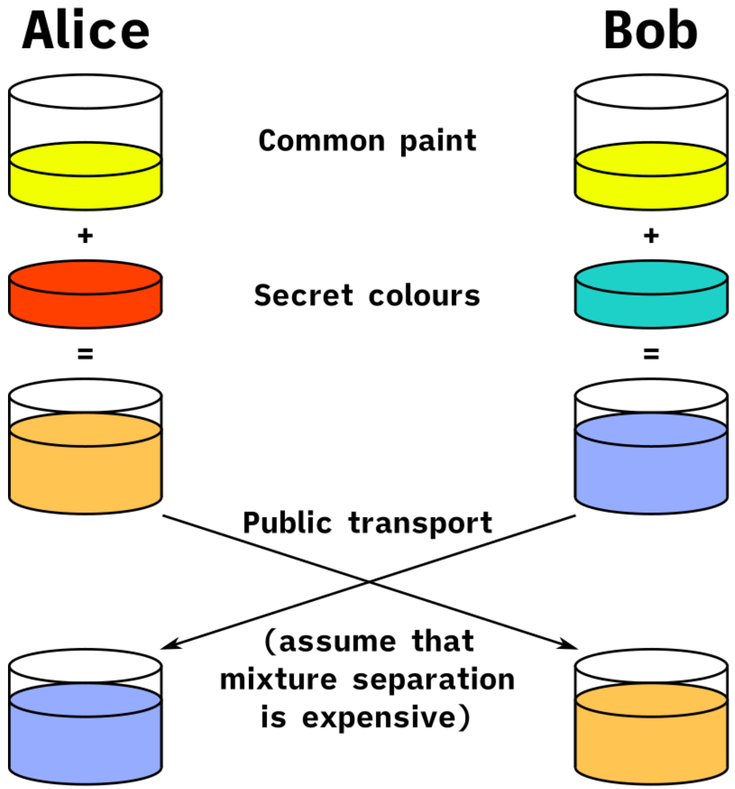
# Diffie-Hellman Key Exchange

# Secure Color Sharing

# Secure Color Sharing

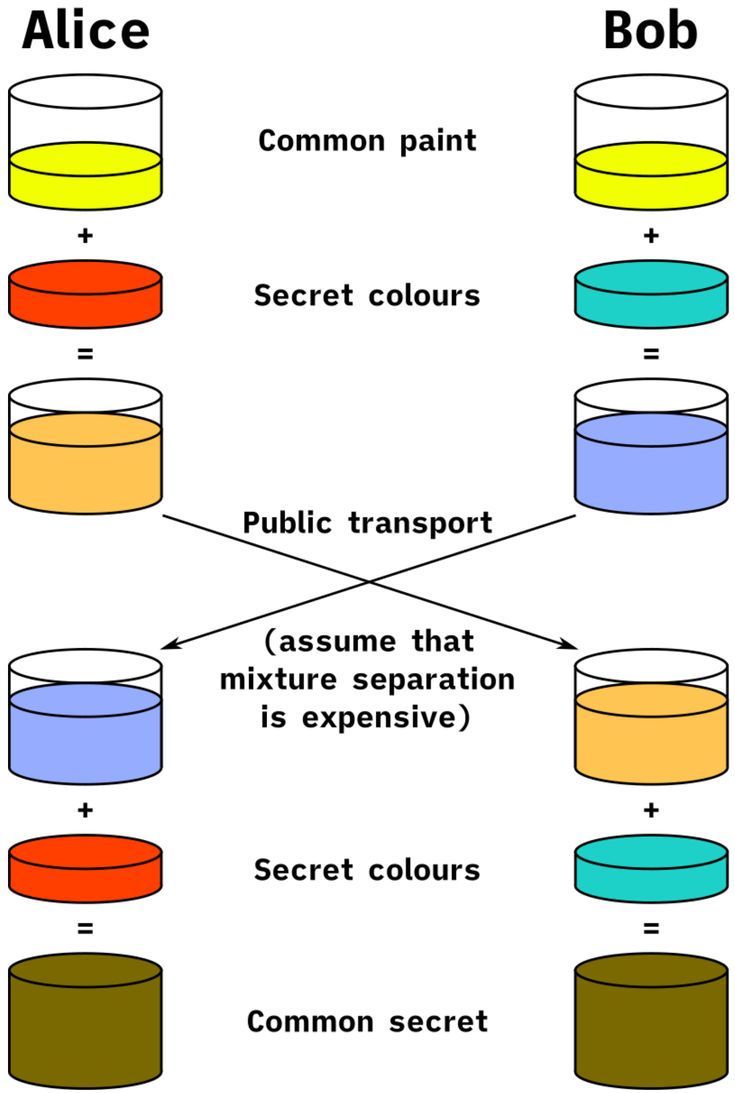


# Secure Color Sharing




Paint assumption: separating a paint mixture is hard

# Secure Color Sharing



# Discrete Log Problem and Diffie-Hellman Problem

- Recall our paint assumption: Separating a paint mixture is hard
  - Is there a mathematical version of this? Yes!
- Assume everyone knows a large prime $p$ (e.g. 2048 bits long) and a generator $g$
  - Don't worry about what a generator is
- **Discrete logarithm problem** (**discrete log problem**): Given $g$, $p$, $g^a \bmod p$ for random $a$, it is computationally hard to find $a$
- **Diffie-Hellman assumption**: Given $g$, $p$, $g^a \bmod p$, and $g^b \bmod p$ for random $a$, $b$, no polynomial time attacker can distinguish between a random value R and $g^{ab} \bmod p$.
  - Intuition: The best known algorithm is to first calculate $a$ and then compute $(g^b)^a \bmod p$, but this requires solving the discrete log problem, which is hard!
  - Note: Multiplying the values doesn't work, since you get $g^{a+b} \bmod p \neq g^{ab} \bmod p$

# Diffie-Hellman Key Exchange

Alice

Public: $g$, $p$

Eve

Bob

Secret key → Generate $a$

Generate $b$

Calculate $g^a \bmod p$

$g^a$

$g^b$

Calculate $g^b \bmod p$

Public key

Receive $g^b \bmod p$

Receive $g^a \bmod p$

Calculate $(g^b)^a \bmod p$

Calculate $(g^a)^b \bmod p$

$a$, $g^a$, $g^b \Rightarrow g^{ab}$

$g^a$, $g^b \not\Rightarrow g^{ab}$

$b$, $g^a$, $g^b \Rightarrow g^{ab}$

Shared symmetric key is $g^{ab}$

# Ephemerality of Diffie-Hellman

- Diffie-Hellman can be used ephemerally (called Diffie-Hellman ephemeral, or DHE)
  - **Ephemeral**: Short-term and temporary, not permanent
  - Alice and Bob discard *a*, *b*, and $K = g^{ab} \bmod p$ when they're done
  - Because you need *a* and *b* to derive *K*, you can never derive *K* again!
  - Sometimes *K* is called a **session key**, because it's only used for a an ephemeral session
- Benefit of DHE: **Forward secrecy**
  - Eve records everything sent over the insecure channel
  - Alice and Bob use DHE to agree on a key $K = g^{ab} \bmod p$
  - Alice and Bob use *K* as a symmetric key
  - After they're done, discard *a*, *b*, and *K*
  - Later, Eve steals all of Alice and Bob's secrets
  - Eve can't decrypt any messages she recorded: Nobody saved *a*, *b*, or *K*, and her recording only has $g^a \bmod p$ and $g^b \bmod p$!

# Diffie-Hellman: the MITM attack

Alice

**Mallory**

Public: $g$, $p$

Bob

Generate $a$

Calculate $g^a$ mod $p$

Generate $m$

Calculate $g^m$ mod $p$

Generate $b$

Calculate $g^b$ mod $p$

Receive $g^m$ mod $p$

Calculate $(g^m)^a$ mod $p$

Receive $g^a$ mod p

Calculate $(g^a)^m$ mod p

Receive $g^b$ mod p

Calculate $(g^b)^m$ mod p

Receive $g^m$ mod $p$

Calculate $(g^m)^b$ mod $p$

$a$, $g^a$, $g^m \Rightarrow g^{am}$

$m$, $g^m$, $g^a \Rightarrow g^{am}$

$m$, $g^m$, $g^b \Rightarrow g^{bm}$

$b$, $g^b$, $g^m \Rightarrow g^{bm}$

# Diffie-Hellman: Issues

- Diffie-Hellman is not secure against a MITM adversary
- DHE is an *active protocol*: Alice and Bob need to be online at the same time to exchange keys
  - What if Bob wants to encrypt something and send it to Alice for her to read later?
  - Secure messaging: How do we use *public-key encryption* to send encrypted messages when Alice and Bob don't share keys and aren't online at the same time?
- Diffie-Hellman does not provide *authentication*
  - You exchanged keys with someone, but Diffie-Hellman makes no guarantees about who you exchanged keys with; it could be Mallory!

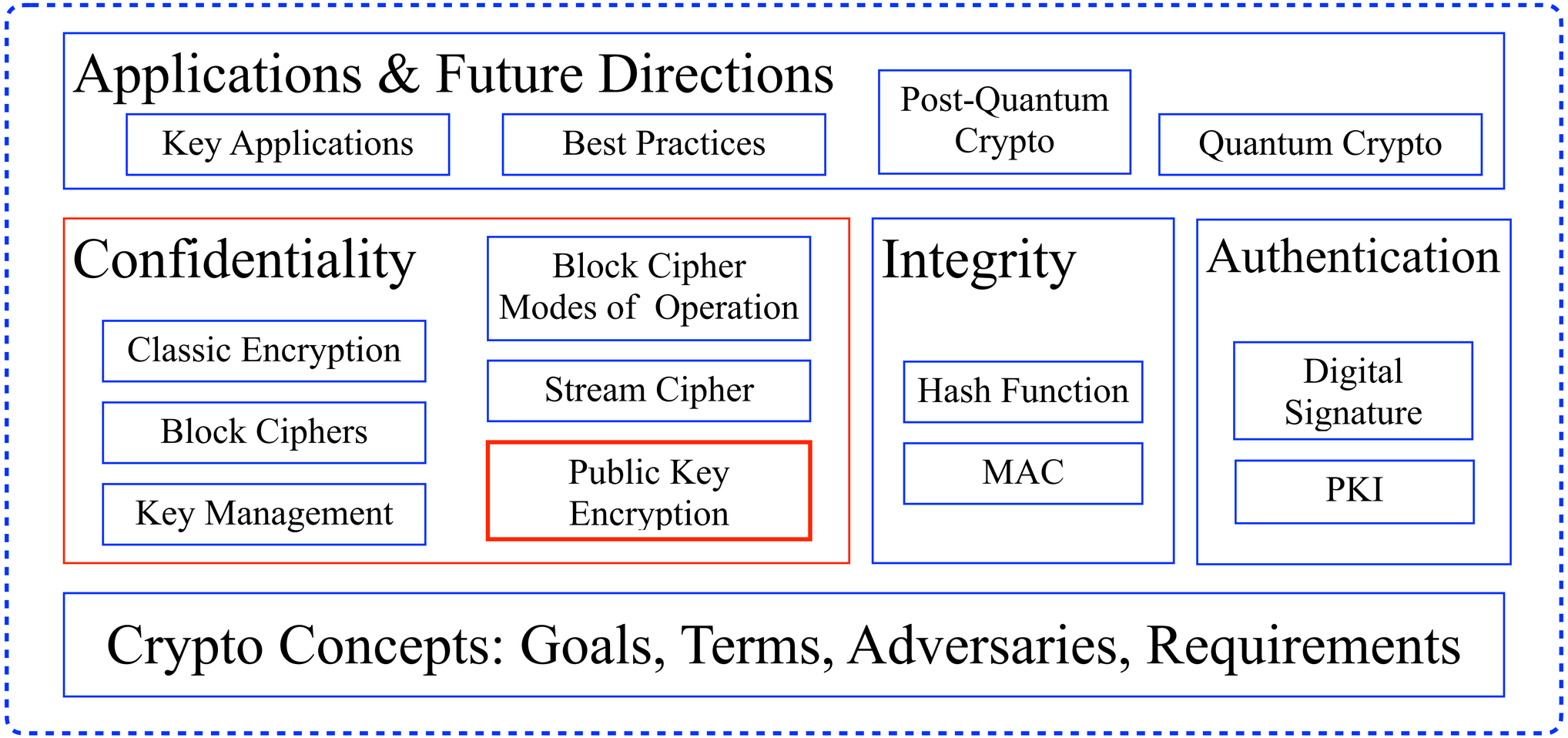
# Elliptic-Curve Diffie-Hellman (ECDH)

- Notice: The discrete-log problem seems hard because exponentiating integers in modular arithmetic “wraps around”
  - Diffie-Hellman can be generalized to any mathematical group that has this cyclic property
  - Discrete-log uses the “multiplicative group of integers mod *p* under generator *g*”
- Elliptic curves: A type of mathematical curve
  - Big idea: Repeatedly adding a point to itself on a curve is another cyclic group
  - You don’t need to understand the math behind elliptic curves
- **Elliptic-curve Diffie-Hellman**: A variation of Diffie-Hellman that uses elliptic curves instead of modular arithmetic
  - Based on the elliptic curve discrete log problem, the analog of the discrete log problem
  - Benefit of ECDH: The underlying problem is harder to solve, so we can use smaller keys (3072-bit DHE is about as secure as 384-bit ECDHE)

# Summary: Diffie-Hellman Key Exchange

- Algorithm:
  - Alice chooses $a$ and sends $g^a \bmod p$ to Bob
  - Bob chooses $b$ and sends $g^b \bmod p$ to Alice
  - Their shared secret is $(g^a)^b = (g^b)^a = g^{ab} \bmod p$
- Diffie-Hellman provides forwards secrecy: Nothing is saved or can be recorded that can ever recover the key
- Diffie-Hellman can be performed over other mathematical groups, such as elliptic-curve Diffie-Hellman (ECDH)
- Issues
  - *Not* secure against MITM
  - Both parties must be online
  - Does not provide authenticity

# The Roadmap of Cryptography



# Public-Key Cryptography

- In public-key schemes, each person has two keys
  - **Public key**: Known to everybody
  - **Private key**: Only known by that person
  - Keys come in pairs: every public key corresponds to one private key
- Uses number theory
  - Examples: Modular arithmetic, factoring, discrete logarithm problem
  - Contrast with symmetric-key cryptography (uses XORs and bit-shifts)
- Messages are numbers
  - Contrast with symmetric-key cryptography (messages are bit strings)
- Benefit: No longer need to assume that Alice and Bob already share a secret
- Drawback: Much slower than symmetric-key cryptography
  - Number theory calculations are much slower than XORs and bit-shifts

# Public-Key Encryption

- Everybody can encrypt with the public key
- Only the recipient can decrypt with the private key

# Public-Key Encryption: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the public key, and *SK* is the private (secret) key
  - Enc(*PK*, *M*) → *C*: Encrypt a plaintext *M* using public key *PK* to produce ciphertext *C*
  - Dec(*SK*, *C*) → *M*: Decrypt a ciphertext *C* using secret key *SK*
- Properties
  - **Correctness**: Decrypting a ciphertext should result in the message that was originally encrypted
    - Dec(*SK*, Enc(*PK*, *M*)) = *M* for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Encryption/decryption should be fast
  - **Security**: Similar to IND-CPA, but Alice (the challenger) just gives Eve (the adversary) the public key, and Eve doesn't request encryptions, except for the pair $M_0$, $M_1$
    - You don't need to worry about this game (it's called "semantic security")

# ElGamal Encryption

# ElGamal Encryption

- Diffie-Hellman key exchange is great: It lets Alice and Bob share a secret over an insecure channel
- Problem: Diffie-Hellman by itself can't send messages. The secret $g^{ab}$ mod $p$ is random.
- Idea: Let's modify Diffie-Hellman so it supports encrypting and decrypting messages directly
- It was proposed in 1985 by Taher Elgamal, an Egyptian cryptographer.
  - The paper: "*Public Key Cryptosystem and A Signature Scheme Based on Discrete Logarithms*"
  - Martin **Hellman** was his PhD dissertation advisor

# ElGamal Encryption: Protocol

- KeyGen():
  - Bob generates private key $b$ and public key $B = g^b \bmod p$
    - Intuition: Bob is completing his half of the Diffie-Hellman exchange
- Enc($B$, $M$):
  - Alice generates a random $r$ and computes $R = g^r \bmod p$
    - Intuition: Alice is completing her half of the Diffie-Hellman exchange
  - Alice computes $M \times B^r \bmod p$
    - Intuition: Alice derives the shared secret and multiples her message by the secret
  - Alice sends $C_1 = R$, $C_2 = M \times B^r \bmod p$
- Dec($b$, $C_1$, $C_2$)
  - Bob computes $C_2 \times C_1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$
    - Intuition: Bob derives the (inverse) shared secret and multiples the ciphertext by the inverse shared secret

# ElGamal Encryption: Security

- Recall Diffie-Hellman problem: Given $g^a \bmod p$ and $g^b \bmod p$, hard to recover $g^{ab} \bmod p$
- ElGamal sends these values over the insecure channel
  - Bob's public key: $B$
  - Ciphertext: $R$, $M \times B^r \bmod p$
- Eve can't derive $g^{br}$, so she can't recover $M$

# ElGamal Encryption: Issues

- Is ElGamal encryption IND-CPA secure?
  - No. The adversary can send $M_0 = 0$, $M_1 \neq 0$
  - Additional padding and other modifications are needed to make it semantically secure
- Malleability: The adversary can tamper with the message
  - The adversary can manipulate $C_1' = C_1$, $C_2' = 2 \times C_2 = 2 \times M \times g^{br}$ to make it look like $2 \times M$ was encrypted

# RSA Encryption

# RSA Encryption: Definition

KeyGen():

- Randomly pick two large primes, $p$ and $q$
  - Done by picking random numbers and then using a test to see if the number is (probably) prime
- Compute $N = pq$
  - $N$ is usually between 2048 bits and 4096 bits long
- Choose $e$
  - Requirement: $e$ is relatively prime to $(p - 1)(q - 1)$
  - Requirement: $2 < e < (p - 1)(q - 1)$
- Compute $d = e^{-1} \bmod (p - 1)(q - 1)$, i.e., $ed \equiv 1 \bmod (p - 1)(q - 1)$
  - Algorithm: Extended Euclid's algorithm
- **Public key**: $N$ and $e$
- **Private key:** $d$

# RSA Encryption: Definition

- Enc($e$, $N$, $M$):
  - Output $M^e \bmod N$
- Dec($d$, $C$):
  - Output $C^d = (M^e)^d \bmod N$

# RSA Encryption: Correctness

1. To prove Theorem: $M^{ed} \equiv M \bmod N$
2. Euler's theorem: $a^{\varphi(N)} \equiv 1 \bmod N$
   - $\varphi(N)$ is the totient function of $N$
   - If $N$ is prime, $\varphi(N) = N - 1$ (Fermat's little theorem)
   - For a semi-prime $pq$, where $p$ and $q$ are prime, $\varphi(pq) = (p - 1)(q - 1)$
   - This is all out-of-scope number theory knowledge
3. Notice: $ed \equiv 1 \bmod (p - 1)(q - 1)$ so $ed \equiv 1 \bmod \varphi(N)$
   - This means that $ed = k\varphi(n) + 1$ for some integer $k$
4. (1) can be written as $M^{k\varphi(N) + 1} \equiv M \bmod N$
5. $M^{k\varphi(N)}M^1 \equiv M \bmod N$
6. $1M^1 \equiv M \bmod N$ by Euler's theorem
7. $M \equiv M \bmod N$

# RSA Encryption: Security

- **RSA problem**: Given $N$ and $C = M^e \bmod N$, it is hard to find $M$
  - No harder than the factoring problem (if you can factor $N$, you can recover $d$)
- Current best solution is to factor $N$, but unknown whether there is an easier way
  - If the RSA problem is as hard as the factoring problem, then the scheme is secure as long as the factoring problem is hard
  - Factoring problem is assumed to be hard (if you don't have a massive quantum computer, that is)

# RSA Encryption: Issues

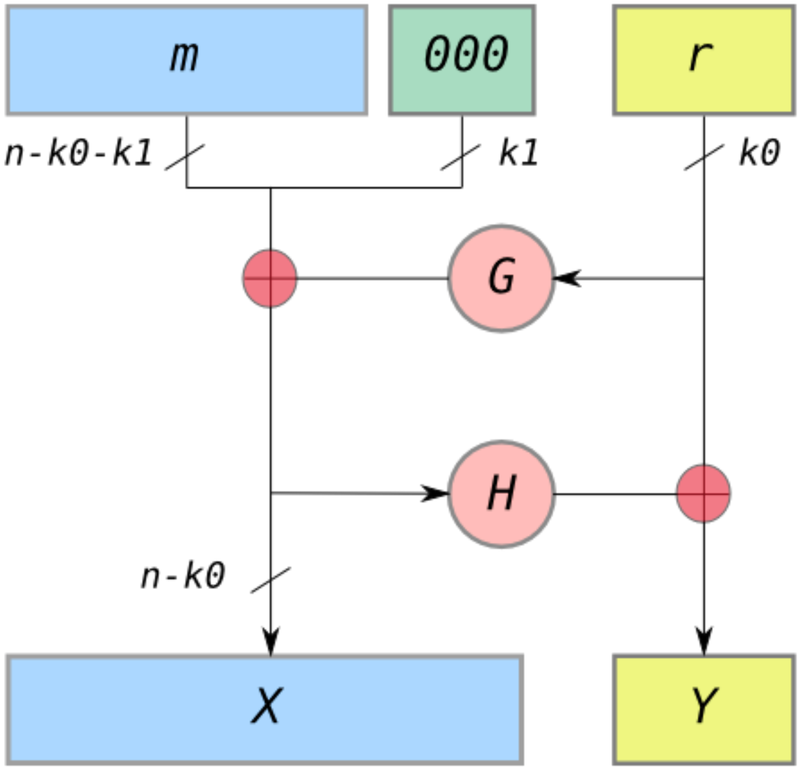
- Is RSA encryption IND-CPA secure?
  - No. It's deterministic. No randomness was used at any point!
- Sending the same message encrypted with different public keys also leaks information
  - $m^{e_a} \bmod N_a$, $m^{e_b} \bmod N_b$, $m^{e_c} \bmod N_c$
  - Small $m$ and $e$ leaks information
    - $e$ is usually small (~16 bits) and often constant (3, 17, 65537)
- Side channel: A poor implementation leaks information
  - The time it takes to decrypt a message depends on the message and the private key
  - This attack has been successfully used to break RSA encryption in OpenSSL
- Result: We need a probabilistic padding scheme

# OAEP

- **Optimal asymmetric encryption padding** (**OAEP**): A variation of RSA that introduces randomness
  - Different from “padding” used for symmetric encryption, used to add randomness instead of dummy bytes
- Idea: RSA can only encrypt “random-looking” numbers, so encrypt the message with a random key

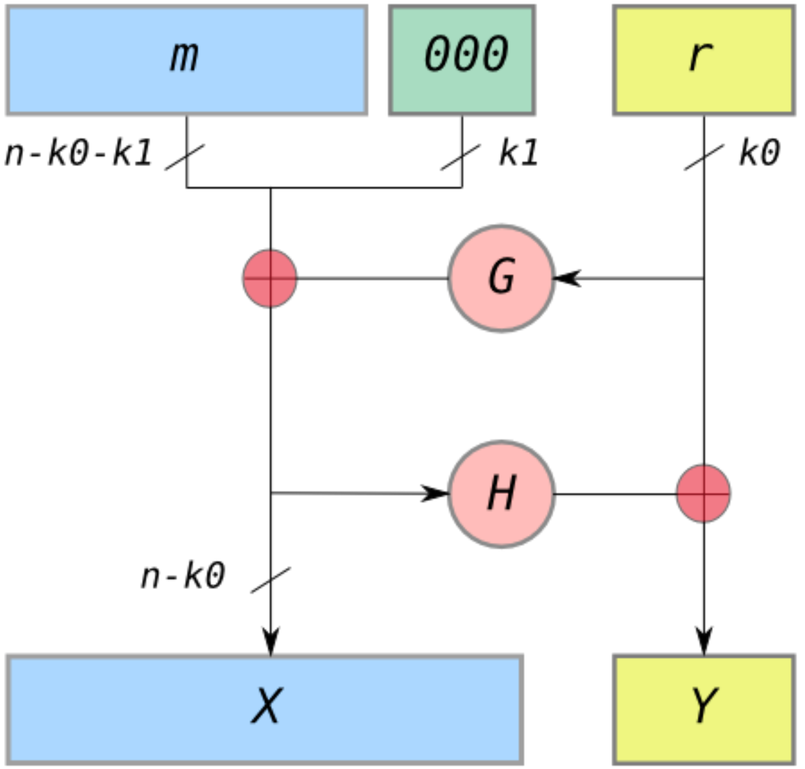
# OAEP: Padding

1. $k_0$ and $k_1$ constants defined in the standard, and $G$ and $H$ are hash functions
   - $M$ can only be $n$ - $k_0$ - $k_1$ bits long
   - $G$ produces a ($n$ - $k_0$)-bit hash, and $H$ produces a $k_0$-bit hash
2. Generate a random, $k_0$-bit string $r$
3. Pad $M$ with $k_1$ 0's
   - Idea: We should see 0's here when unpadding, or else someone tampered with the message
4. Compute $X = (M \,||\, 00\ldots0) \oplus G(r)$
5. Compute $Y = r \oplus H(X)$
6. Result: $X \,||\, Y$



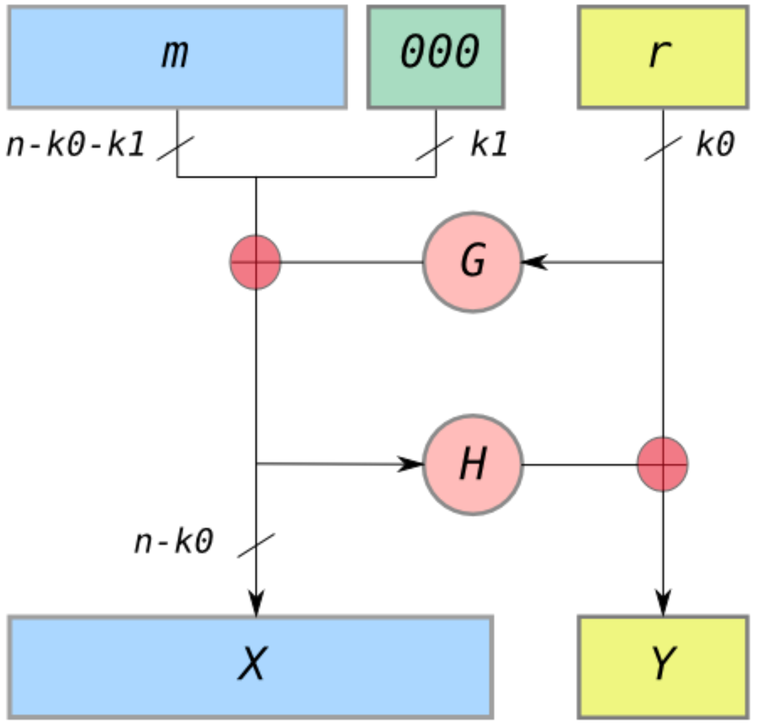
# OAEP: Unpadding

1. Compute $r = Y \oplus H(X)$
2. Compute $M \,||\, 00...0 = X \oplus G(r)$
3. Verify that $M \,||\, 00...0$ actually ends in $k_1$ 0's
   - Error if not



# OAEP

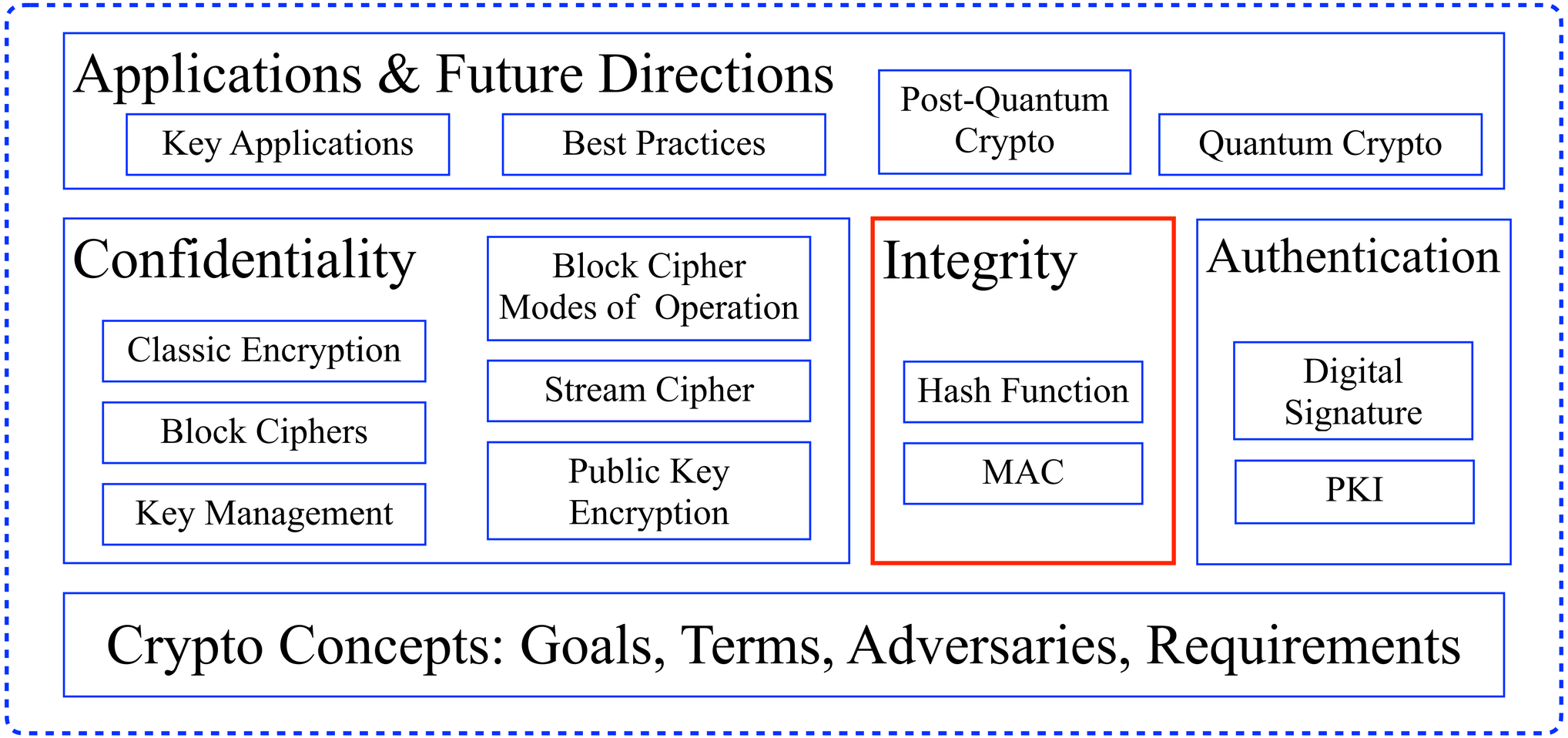
- Even though *G* and *H* are irreversible, we can recover their inputs using XOR and work backwards
- This structure is called a **Feistel network**
  - Can be used for encryption algorithms if *G* and *H* depend on a key
    - Example: DES
- It converts a **deterministic** encryption scheme (e.g., traditional RSA) into a **probabilistic** scheme
- **Takeaway**: To fix the problems with RSA (it's only secure encrypting random numbers and isn't IND-CPA), use RSA with OAEP, abbreviated as RSA-OAEP



# Hybrid Encryption

- Issues with public-key encryption
  - Notice: We can only encrypt small messages because of the modulo operator
  - Notice: There is a lot of math, and computers are slow at math
  - Result: Asymmetric doesn't work for large messages
- **Hybrid encryption**: Encrypt data under a randomly generated key *K* using symmetric encryption, and encrypt *K* using asymmetric encryption
  - Benefit: Now we can encrypt large amounts of data quickly using symmetric encryption, and we still have the security of asymmetric encryption
- Almost all cryptographic systems use hybrid encryption

# The Roadmap of Cryptography



# Cryptographic Hash Function: Definition

- Hash function: $H(M)$
  - Input: *Arbitrary* length message $M$
  - Output: *Fixed* length, $n$-bit hash
  - Sometimes written as $\{0, 1\}^* \rightarrow \{0, 1\}^n$
- Properties
  - **Correctness**: Deterministic
    - Hashing the same input always produces the same output
  - **Efficiency**: Efficient to compute
  - **Security**: One-way-ness (“preimage resistance”)
  - **Security**: Collision-resistance
  - **Security:** Random/unpredictability, no predictable patterns for how changing the input affects the output
    - Changing 1 bit in the input causes the output to be completely different
    - Also called “random oracle” assumption

# Hash Function: Intuition

- A hash function provides a fixed-length “fingerprint” over a sequence of bits
- Example: Document comparison
  - If Alice and Bob both have a 1 GB document, they can both compute a hash over the document and (securely) communicate the hashes to each other
  - If the hashes are the same, the files must be the same, since they have the same “fingerprint”
  - If the hashes are different, the files must be different

# Hash Function: One-way-ness or Preimage Resistance

- **Informal:** Given an output *y*, it is infeasible to find *any* input *x* such that *H*(*x*) = *y*
- **More formally:** For all polynomial time adversary, **Probability[**

  *x* chosen randomly from plaintext space;

  *y* = *H*(*x*): Adv(*y*) outputs *x′* s.t. *H*(*x′*) = *y*

  ] is negligible
- Intuition: Here's an output. Can you find an input that hashes to this output?
  - Note: The adversary just needs to find *any* input, not necessarily the input that was actually used to generate the hash
- Example: Is H(x) = 1 one-way?
  - No, because given output 1, an attacker can return any number x

# Hash Function: Collision Resistance

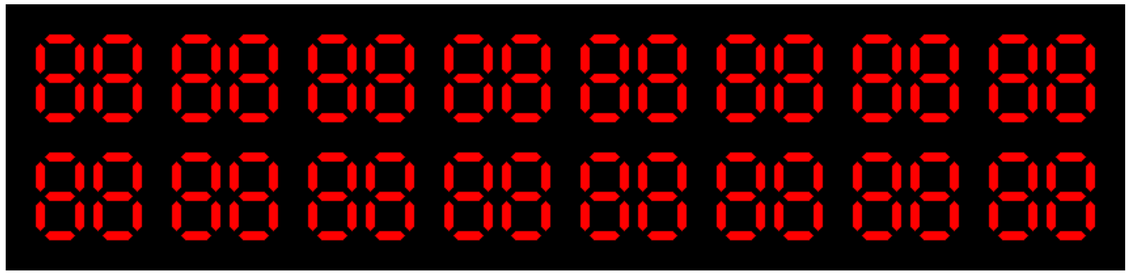
- **Collision**: Two different inputs with the same output
  - $x \neq x'$ and $H(x) = H(x')$
  - Can we design a hash function with no collisions?
    - No, because there are more inputs than outputs (pigeonhole principle)
  - However, we want to make finding collisions *infeasible* for an attacker
- **Collision resistance**: It is infeasible to (i.e. no polynomial time attacker can) find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$
- Intuition: Can you find *any* two inputs that collide with the same hash output for *any* output?

# Hash Function: Collision Resistance

- **Birthday attack**: Finding a collision on an *n*-bit output requires only $2^{n/2}$ tries on average
  - This is why a group of 23 people are >50% likely to have at least one birthday in common

# Hash Function: Examples

- MD5
  - Output: 128 bits
  - Security: Completely broken
- SHA-1
  - Output: 160 bits
  - Security: Completely broken in 2017
  - Was known to be weak before 2017, but still used sometimes
- SHA-2
  - Output: 256, 384, or 512 bits (sometimes labeled SHA-256, SHA-384, SHA-512)
  - Not currently broken, but some variants are vulnerable to a length extension attack
  - Current standard
- SHA-3 (Keccak)
  - Output: 256, 384, or 512 bits
  - Current standard (not meant to replace SHA-2, just a different construction)



A GIF that displays its own MD5 hash

# Length Extension Attacks

- **Length extension attack**: Given $H(x)$ and the length of $x$, but not $x$, an attacker can create $H(x \,||\, m)$ for any $m$ of the attacker's choosing
  - Note: This doesn't violate any property of hash functions but is undesirable in some circumstances, e.g., message authentication code
  - All hash functions based on the Merkle–Damgård construction are vulnerable to this attack: MD5, SHA-1, and SHA-2
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 and HMAC are not vulnerable

# Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Mozilla publishes a new version of Firefox on some download servers
  - Alice downloads the program binary
  - How can she be sure that nobody tampered with the program?
- Idea: use cryptographic hashes
  - Mozilla hashes the program binary and publishes the hash on its website
  - Alice hashes the binary she downloaded and checks that it matches the hash on the website
  - If Alice downloaded a malicious program, the hash would not match (tampering detected!)
  - An attacker can’t create a malicious program with the same hash (collision resistance)
- Threat model: We assume the attacker cannot modify the hash on the website
  - We have integrity, as long as we can communicate the hash securely

# Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Alice and Bob want to communicate over an insecure channel
  - Mallory might tamper with messages
- Idea: Use cryptographic hashes
  - Alice sends her message with a cryptographic hash over the channel
  - Bob receives the message and computes a hash on the message
  - Bob checks that the hash he computed matches the hash sent by Alice
- Threat model: Mallory can modify the message *and the hash*
  - No integrity!

# Do hashes provide integrity?

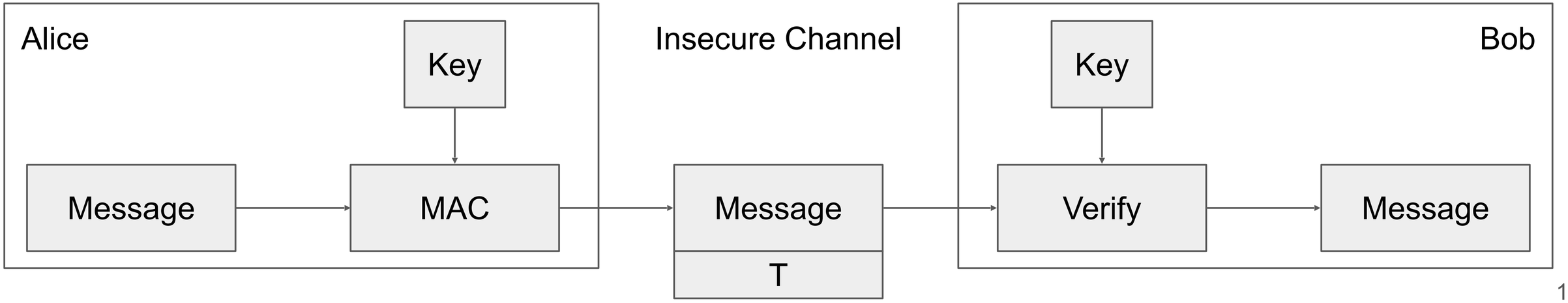
- It depends on your threat model
- If the attacker can modify the hash, hashes don’t provide integrity
- Main issue: Hashes are *unkeyed* functions
  - There is no secret key being used as input, so any attacker can compute a hash on any value
- Next: Use hashes to design schemes that provide integrity

# How to Provide Integrity

- Reminder: We're still in the symmetric-key setting
  - Assume that Alice and Bob share a secret key, and attackers don't know the key
- We want to attach some piece of information to *prove* that someone with the key sent this message
  - This piece of information can only be generated by someone with the key

# MACs: Usage

- Alice wants to send *M* to Bob, but doesn't want Mallory to tamper with it
- Alice sends *M* and *T* = MAC(*K*, *M*) to Bob
- Bob receives *M* and *T*
- Bob computes MAC(*K*, *M*) and checks that it matches *T*
- If the MACs match, Bob is confident the message has not been tampered with (integrity)



# MACs: Definition

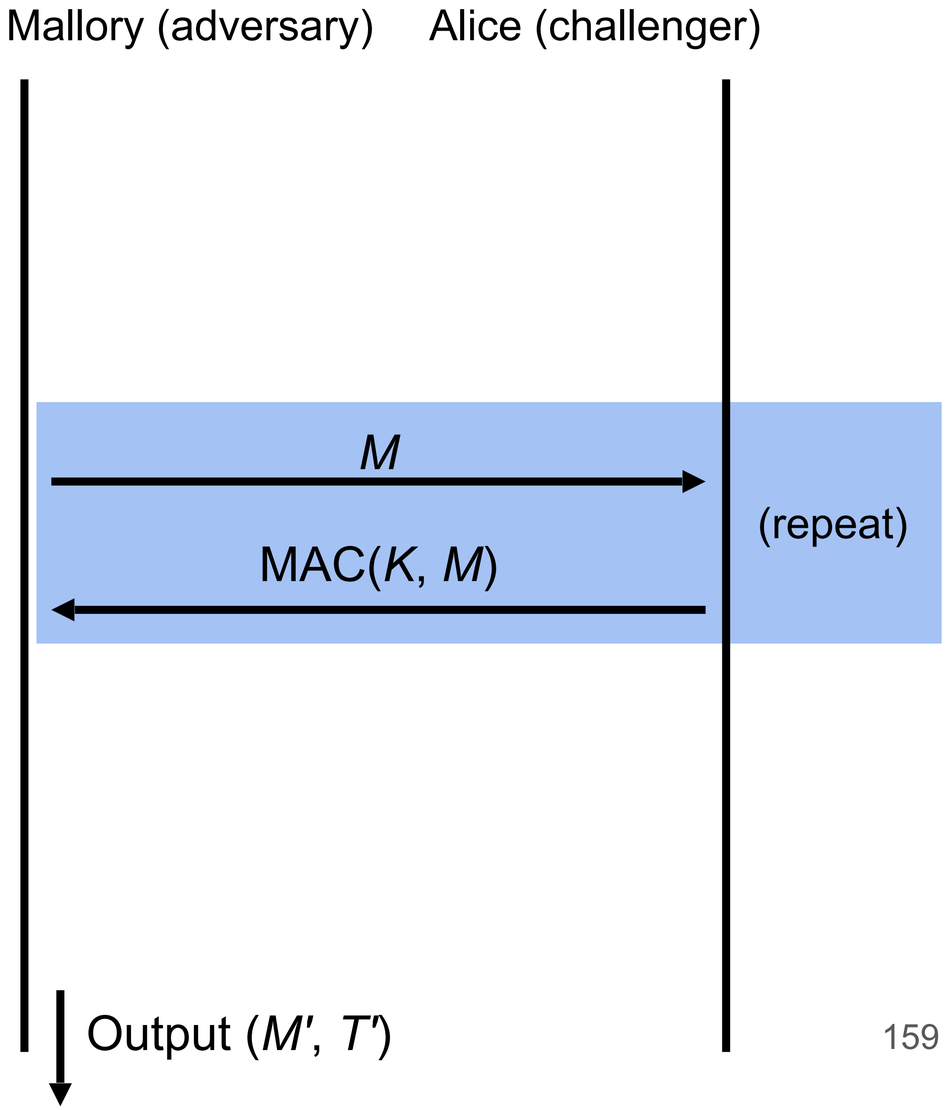
- Two parts:
  - KeyGen() → *K*: Generate a key *K*
  - MAC(*K*, *M*) → *T*: Generate a tag *T* for the message *M* using key *K*
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(*K*, *M*, *T*) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: EU-CPA (existentially unforgeable under chosen plaintext attack)

# Defining Integrity: EU-CPA

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
  - Mallory cannot generate MAC($K$, $M'$) without $K$
  - Mallory cannot find any $M' \neq M$ such that MAC($K$, $M'$) = MAC($K$, $M$)
- Formally defined by a security game: existential unforgeability under chosen-plaintext attack, or EU-CPA
- MACs should be unforgeable under chosen plaintext attack
  - Intuition: Like IND-CPA, but for integrity and authenticity
  - Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before

# Defining Integrity: EU-CPA

1. Mallory may send messages to Alice and receive their tags
2. Eventually, Mallory creates a message-tag pair ($M'$, $T'$)
   - $M'$ cannot be a message that Mallory requested earlier
   - If $T'$ is a valid tag for $M'$, then Mallory wins. Otherwise, she loses.
   - A scheme is EU-CPA secure if for *all* polynomial time adversaries, the probability of winning is 0 or negligible



# Example: NMAC

- Can we use secure cryptographic hashes to build a secure MAC?
  - Intuition: Hash output is unpredictable and looks random, so let's hash the key and the message together
- KeyGen():
  - Output two random, $n$-bit keys $K_1$ and $K_2$, where $n$ is the length of the hash output
- NMAC($K_1$, $K_2$, $M$):
  - Output $H(K_1 \,||\, H(K_2 \,||\, M))$
- NMAC is EU-CPA secure if the two keys are different
  - Provably secure if the underlying hash function is secure
- Intuition: Using two hashes prevents a length extension attack
  - Otherwise, an attacker who sees a tag for $M$ could generate a tag for $M \,||\, M'$

# Example: HMAC

- Issues with NMAC:
  - Recall: NMAC($K_1$, $K_2$, $M$) = $H(K_1 \,||\, H\,(K_2 \,||\, M))$
  - We need two different keys
  - NMAC requires the keys to be the same length as the hash output ($n$ bits)
  - Can we use NMAC to design a scheme that uses one key?
- HMAC($K$, $M$):
  - Compute $K'$ as a version of $K$ that is the length of the hash output
    - If $K$ is too short, pad $K$ with 0's to make it $n$ bits (be careful with keys that are too short and lack randomness)
    - If $K$ is too long, hash it so it's $n$ bits
  - Output $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$

# Example: HMAC

- HMAC(*K*, *M*):
  - Compute *K*' as a version of *K* that is the length of the hash output
    - If *K* is too short, pad *K* with 0's to make it *n* bits (be careful with keys that are too short and lack randomness)
    - If *K* is too long, hash it so it's *n* bits
  - Output $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$
- Use *K*' to derive two different keys
  - *opad* (outer pad) is the hard-coded byte `0x5c` repeated until it's the same length as *K*'
  - *ipad* (inner pad) is the hard-coded byte `0x36` repeated until it's the same length as *K*'
  - As long as *opad* and *ipad* are different, you'll get two different keys
  - For paranoia, the designers chose two very different bit patterns, even though they theoretically need only differ in one bit

# HMAC Properties

- HMAC($K$, $M$) = $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$
- HMAC is a hash function, so it has the properties of the underlying hash too
  - It is collision resistant
  - Given HMAC($K$, $M$) and $K$, an attacker can't learn $M$
  - If the underlying hash is secure, HMAC doesn't reveal $M$, but it is still deterministic
- You can't verify a tag $T$ if you don't have $K$
  - This means that an attacker can't brute-force the message $M$ without knowing $K$

# Do MACs provide integrity?

- Do MACs provide integrity?
  - Yes. An attacker cannot tamper with the message without being detected
- Do MACs provide authenticity?
  - It depends on your threat model
  - If a message has a valid MAC, you can be sure it came from *someone with the secret key*, but you can't narrow it down to one person
  - If only two people have the secret key, MACs provide authenticity: it has a valid MAC, and it's not from me, so it must be from the other person
- Do MACs provide confidentiality?
  - MACs are deterministic ⇒ No IND-CPA security
  - MACs in general have no confidentiality guarantees; they can leak information about the message
    - HMAC doesn't leak information about the message, but it's still deterministic, so it's not IND-CPA secure

# Authenticated Encryption

# Authenticated Encryption: Definition

- **Authenticated encryption** (**AE**): A scheme that simultaneously guarantees confidentiality and integrity (and authenticity, depending on your threat model) on a message
- Two ways of achieving authenticated encryption:
  - Combine schemes that provide confidentiality with schemes that provide integrity
  - Use a scheme that is designed to provide confidentiality and integrity

# Combining Schemes: Let's design it together

- You can use:
  - An IND-CPA encryption scheme (e.g. AES-CBC): Enc($K$, $M$) and Dec($K$, $M$)
  - An unforgeable MAC scheme (e.g. HMAC): MAC($K$, $M$)
- First attempt: Alice sends Enc($K_1$, $M$) and MAC($K_2$, $M$)
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? No, the MAC is not IND-CPA secure
- Idea: Let's compute the MAC on the *ciphertext* instead of the plaintext: Enc($K_1$, $M$) and MAC($k_2$, Enc($K_1$, $M$))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, the MAC might leak info about the ciphertext, but that's okay
- Idea: Let's encrypt the MAC too: Enc($K_1$, $M$ || MAC($K_2$, $M$))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, everything is encrypted

# MAC-then-Encrypt or Encrypt-then-MAC?

- MAC-then-encrypt
  - First compute MAC($K_2$, $M$)
  - Then encrypt the message and the MAC together: Enc($K_1$, $M$ || MAC($K_2$, $M$))
- Encrypt-then-MAC
  - First compute Enc($K_1$, $M$)
  - Then MAC the ciphertext: MAC($K_2$, Enc($K_1$, $M$))
- Which is better?
  - In theory, both are IND-CPA and EU-CPA secure if applied properly
  - MAC-then-encrypt has a flaw: You don't know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it's more robust to mistakes

# Key Reuse

- **Key reuse**: Using the same key in two different use cases
  - Note: Using the same key multiple times for the same use (e.g. computing HMACs on different messages in the same context with the same key) is not key reuse
- Reusing keys can cause the underlying algorithms to interfere with each other and affect security guarantees
  - Example: If you use a block-cipher-based MAC algorithm and a block cipher chaining mode, the underlying block ciphers may no longer be secure
  - Thinking about these attacks is hard

# Key Reuse

- Simplest solution: Do not reuse keys! One key per *use*.
  - Encrypt a piece of data and MAC a piece of data?
    - Different use; different key
  - MAC one of Alice's messages to Bob and MAC one of Bob's messages to Alice?
    - Different use; different key
  - Encrypt one of Alice's files and encrypt another one of Alice's files?
    - It's *probably* fine to use the same key

# TLS 1.0 “Lucky 13” Attack

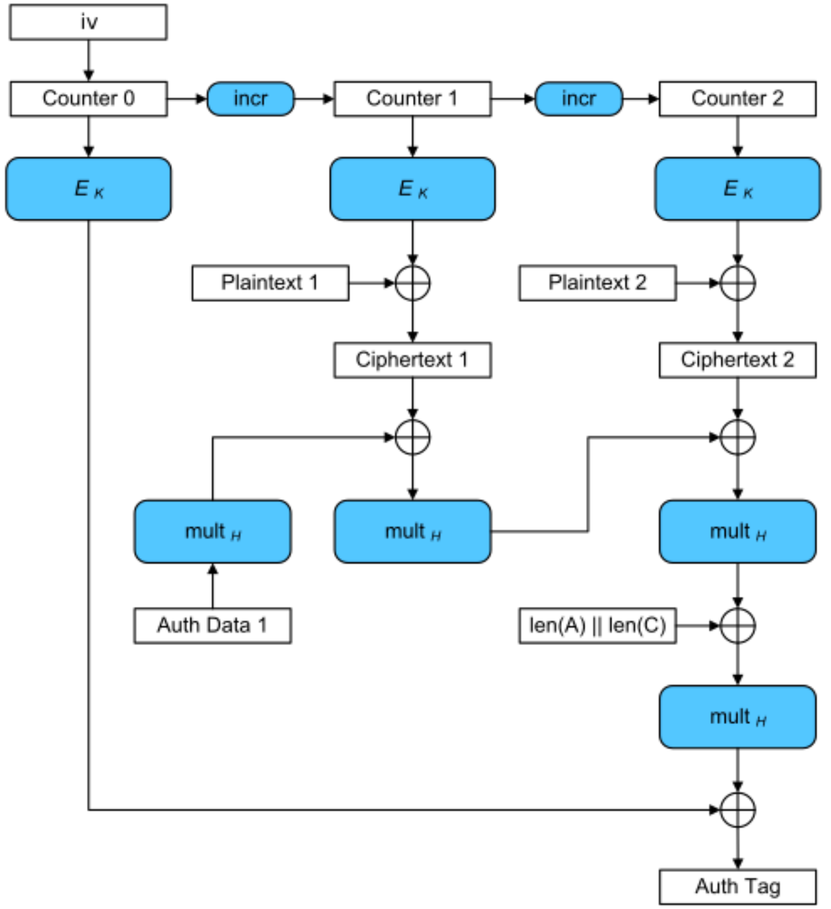
- TLS: A protocol for sending encrypted and authenticated messages over the Internet (we’ll study it more in the networking unit)
- TLS 1.0 uses MAC-then-encrypt: Enc($K_1$, $M$ || MAC($K_2$, $M$))
  - The encryption algorithm is AES-CBC
- The Lucky 13 attack abuses MAC-then-encrypt to read encrypted messages
  - Guess a byte of plaintext and change the ciphertext accordingly
  - The MAC will error, but the time it takes to error is different depending on if the guess is correct
  - Attacker measures how long it takes to error in order to learn information about plaintext
  - TLS will send the message again if the MAC errors, so the attacker can guess repeatedly
- Takeaways
  - Side channel attack: The algorithm is proved secure, but poor implementation made it vulnerable
  - Always encrypt-then-MAC

# AEAD Encryption

- Second method for authenticated encryption: Use a scheme that is designed to provide confidentiality, integrity, and authenticity
- **Authenticated encryption with additional data** (**AEAD**): An algorithm that provides both confidentiality and integrity over the plaintext and integrity over *additional data*
  - Additional data is usually context (e.g. memory address), so you can't change the context without breaking the MAC
- Great if used correctly: No more worrying about MAC-then-encrypt
  - If you use AEAD incorrectly, you lose *both* confidentiality and integrity/authentication
  - Example of correct usage: Using a crypto library with AEAD

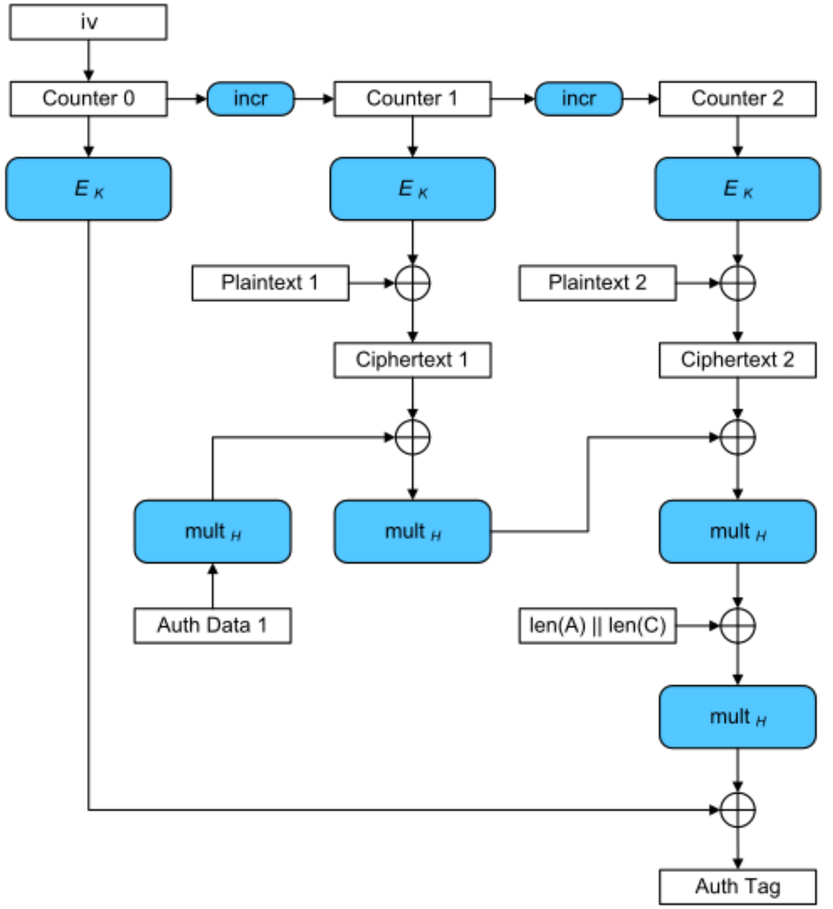
# AEAD Example: Galois Counter Mode (GCM)

- **Galois Counter Mode** (**GCM**): An AEAD block cipher mode of operation
- $E_K$ is standard block cipher encryption
- $\text{mult}_H$ is 128-bit multiplication over a special field (Galois multiplication)
  - Don’t worry about the math



# AEAD Example: Galois Counter Mode (GCM)

- Very fast mode of operation
  - Fully parallel encryption
  - Galois multiplication isn't parallelizable, but it's very fast
- Drawbacks
  - IV reuse leads to loss of confidentiality, integrity, and authentication
  - This wouldn't happen if you used AES-CTR and HMAC-SHA256
  - Implementing Galois implementation is difficult and easy to screw up
- **Takeaway**: GCM provides integrity and confidentiality, but if you misuse it, it's even worse than CTR mode



# Hashes: Summary

- Map arbitrary-length input to fixed-length output
- Output is deterministic and unpredictable
- Security properties
  - One way: Given an output $y$, it is infeasible to find any input $x$ such that $H(x) = y$.
  - Second preimage resistant: Given an input $x$, it is infeasible to find another input $x' \neq x$ such that $H(x) = H(x')$.
  - Collision resistant: It is infeasible to find another any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.
- Some hashes are vulnerable to length extension attacks
- Hashes don’t provide integrity (unless you can publish the hash securely)

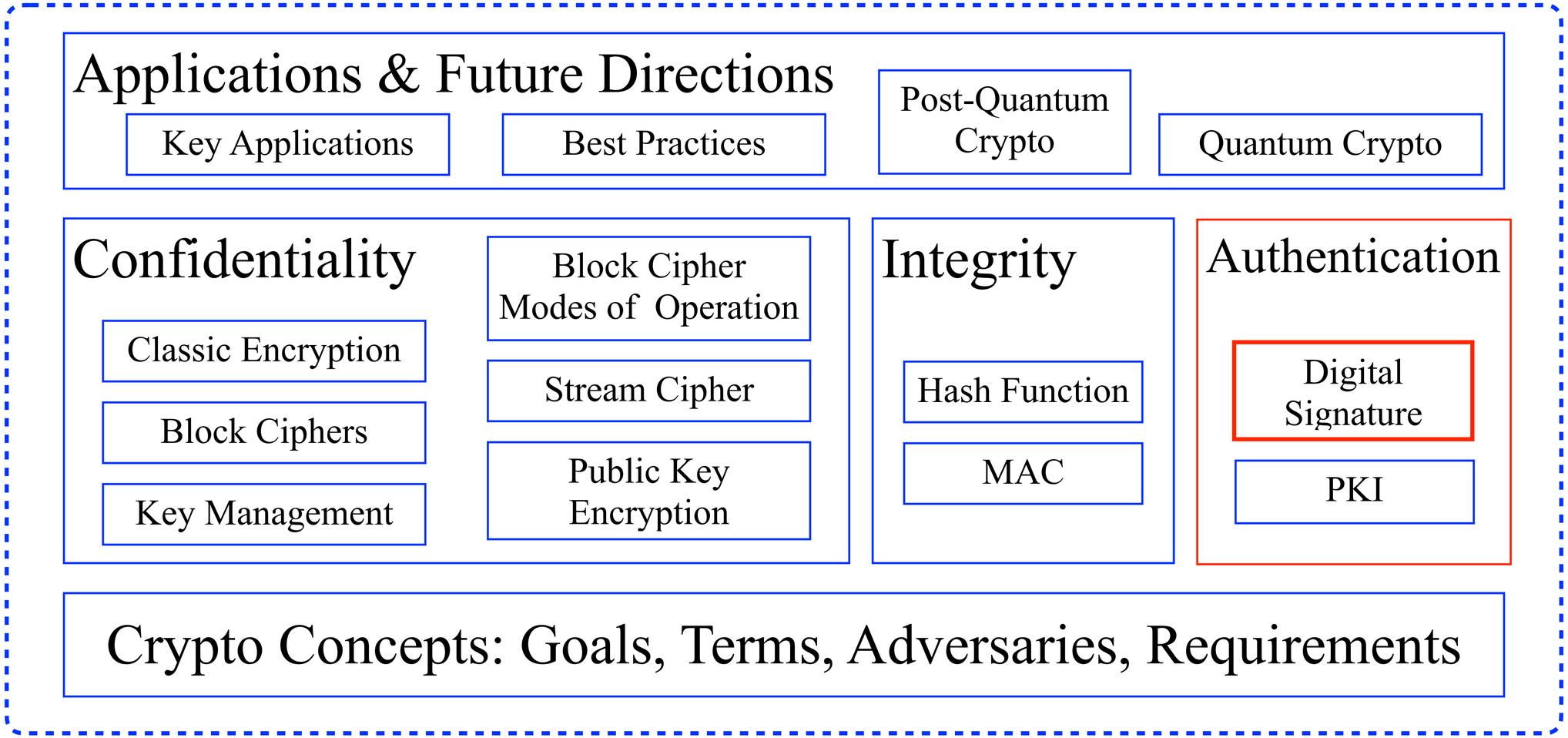
# MACs: Summary

- Inputs: a secret key and a message
- Output: a tag on the message
- A secure MAC is unforgeable: Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before
  - Example: HMAC($K$, $M$) = $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$
- MACs do not provide confidentiality

# Authenticated Encryption: Summary

- Authenticated encryption: A scheme that simultaneously guarantees confidentiality and integrity (and authenticity) on a message
- First approach: Combine schemes that provide confidentiality with schemes that provide integrity and authenticity
  - MAC-then-encrypt: Enc($K_1$, $M$ || MAC($K_2$, $M$))
  - Encrypt-then-MAC: Enc($K_1$, $M$) || MAC($K_2$, Enc($K_1$, $M$))
  - Always use Encrypt-then-MAC because it's more robust to mistakes
- Second approach: Use AEAD encryption modes designed to provide confidentiality, integrity, and authenticity
  - Drawback: Incorrectly using AEAD modes leads to losing *both* confidentiality and integrity/ authentication

# The Roadmap of Cryptography



# Digital Signatures

- Asymmetric cryptography is good because we don’t need to share a secret key
- Digital signatures are the asymmetric way of providing integrity/authenticity to data
- Assume that Alice and Bob can communicate public keys without Mallory interfering
  - We will see how to fix this limitation later

# Public-key Signatures

- Only the owner of the private key can sign messages with the private key
- Everybody can verify the signature with the public key


# Digital Signatures: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify the signature *sig* on message *M* using the verify key *PK* and output 1 if valid and 0 if invalid
- Properties
  - **Correctness**: Verification should be successful for a signature generated over any message
    - Verify(*PK*, *M*, Sign(*SK*, *M*)) = 1 for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Signing/verifying should be fast
  - **Security**: EU-CPA, same as for MACs

# Digital Signatures in Practice

- If you want to sign message *M*:
  - First hash *M*
  - Then sign H(*M*)
- Why do digital signatures use a hash?
  - Allows signing arbitrarily long messages
- Digital signatures provide integrity *and authenticity* for *M*
  - The digital signature acts as proof that the private key holder signed H(*M*), so you know that *M* is authentically endorsed by the private key holder

# RSA Signatures

# RSA Signatures

- Recall RSA encryption: $M^{ed} \equiv M \bmod N$
  - There is nothing special about using *e* first or using *d* first!
  - If we encrypt using *d*, then anyone can “decrypt” using *e*
    - Given *x* and $x^d \bmod N$, can’t recover *d* because of discrete-log problem, so *d* is safe

# RSA Signatures: Definition

- KeyGen():
  - Same as RSA encryption:
    - **Public key**: $N$ and $e$
    - **Private key:** $d$
- Sign($d$, $M$):
  - Compute $H(M)^d \bmod N$
- Verify($e$, $N$, $M$, $sig$)
  - Verify that $H(M) \equiv sig^e \bmod N$

# DSA Signatures

# DSA Signatures

- A signature scheme based on Diffie-Hellman
- Usage
  - Alice generates a public-private key pair and publishes her public key
  - To sign a message, Alice generates a random, secret value *k* and does some computation
  - Note: *k* is not Alice's private key, but a per-message key
  - Note: *k* is sometimes called a nonce but it is not: it must be *random* and never reused
  - The signature itself does not include *k.*
- *k* must be random and secret for each message
  - An attacker who learns *k* can also learn Alice's private key, since k and the private key are jointly used to generate the signature
  - If Alice reuses *k* on two signatures, an attacker can learn *k* (and use *k* to learn her private key)

# DSA Signatures: Attacks

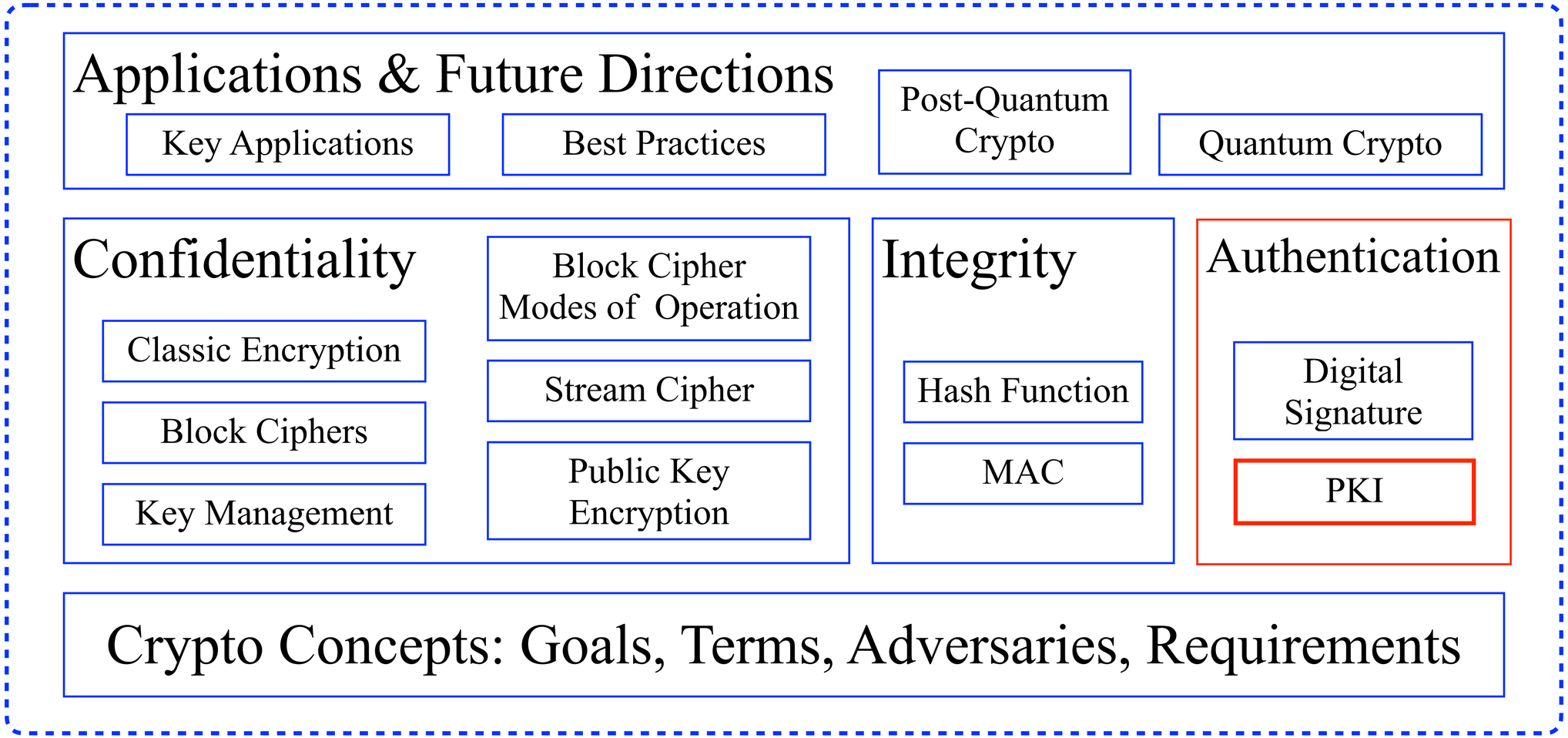
- Android OS vulnerability (2013)
  - The "SecureRandom" function in its random number generator (RNG) wasn't actually secure!
  - Not only was it low entropy, it would sometimes return the same value multiple times
- Multiple Bitcoin wallet apps on Android were affected
  - Bitcoin payments are signed with elliptic-curve DSA and published publicly
  - Insecure RNG caused multiple payments to be signed with the same *k*
- Attack: Someone scanned for all Bitcoin transactions signed insecurely
  - Recall: When multiple signatures use the same *k*, the attacker can learn *k* and the private key
  - In Bitcoin, your private key unlocks access to all your money

https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html

# DSA Signatures: Attacks

- Chromebooks have a built-in U2F (universal second factor) security key
  - Uses signatures to let the user log in to particular websites
  - Signature algorithm: 256-bit elliptic-curve DSA
- There was a bug in the secure hardware!
  - Instead of using 256-bit *k*, a bug caused *k* to be 32 bits long!
  - An attacker with a signature could simply try all possible values of *k*
- Fortunately the damage was slight
  - Each signature is only valid for logging into a single website
  - Each website used its own private key
- **Takeaway**: DSA (or ECDSA) is particularly vulnerable to incorrect implementations, compared with RSA signatures

# The Roadmap of Cryptography



# Certificates

# Review: Public-Key Cryptography

- Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key
- What's the catch?

# Problem: Distributing Public Keys

- Public-key cryptography alone is not secure against man-in-the-middle attacks
- Scenario
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- What can Mallory do?
  - Replace Bob's public key with Mallory’s public key
  - Now Alice has encrypted the message with Mallory’s public key, and Mallory can read it!

# Problem: Distributing Public Keys

Alice

Mallory

Bob

Generate $PK_B$, $SK_B$

Send $PK_B$

Send $PK_M$

Receive $PK_M$

Send $\{M\}_{PK_M}$

Decrypt $\{M\}_{PK_M}$

Send $\{M\}_{PK_B}$

Decrypt $\{M\}_{PK_B}$

# Problem: Distributing Public Keys

- Idea: Sign Bob’s public key to prevent tampering
- Problem
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob’s public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- You cannot gain trust if you trust nothing. You need a root of trust!
  - **Trust anchor**: Someone that we implicitly trust
  - From our trust anchor, we can begin to trust others

# Trust-on-First-Use

- **Trust-on-first-use**: The first time you communicate, trust the public key that is used and warn the user if it changes in the future
  - Used in SSH and a couple other protocols
  - Idea: Attacks aren’t frequent, so assume that you aren’t being attacked the first time communicate
  - Also known as “**Leap of Faith**”

# Certificates

- **Certificate**: A signed endorsement of someone's public key
  - A certificate contains at least two things: The **identity** of the person, and the **key**
- Abbreviated notation
  - Encryption under a public key *PK*: {"Message"}$_{PK}$
  - Signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't send the signature by itself!
- Scenario: Alice wants Bob's public key. Alice trusts EvanBot ($PK_E$, $SK_E$)
  - EvanBot is our trust anchor
  - If we trust $PK_E$, a certificate we would trust is {"Bob's public key is $PK_B$"}$_{SK_E^{-1}}$

# Attempt #1: The Trusted Directory

- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
  - The TD has a public/private keypair $PK_{TD}$, $SK_{TD}$
  - The directory publishes $PK_{TD}$ so that everyone knows it (baked into computers, phones, OS, etc.)
  - When you request Bob's public key, the directory sends a certificate for Bob's public key
    - {"Bob's public key is $PK_B$"}$_{SK_{TD}^{-1}}$
  - If you trust the directory, then now you trust every public key from the directory
- What do we have to trust?
  - We have received TD's key correctly
  - TD won't sign a key without verifying the identity of the owner

# Attempt #1: The Trusted Directory

- Problems: Scalability
  - One directory won’t have enough compute power to serve the entire world
- Problem: Single point of failure
  - If the directory fails, *cryptography stops working*
  - If the directory is compromised, you can’t trust anyone
  - If the directory is compromised, it is difficult to recover

# Certificate Authorities

- Addressing scalability: Hierarchical trust
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {“Xianghang’s public key is $PK_{xh}$, and I trust her to sign for USTC”}$_{SK_{root}^{-1}}$
    - {“Alice’s public key is $PK_{al}$, and I trust her to sign for the CS department”}$_{SK_{xh}^{-1}}$
    - {“Bob’s public key is $PK_{bb}$ (but I don’t trust him to sign for anyone else)”}$_{SK_{al}^{-1}}$
  - There still will be a root of trust (**root certificate authority**, or **root CA**)
  - Xianghang and Alice receive delegated trust (**intermediate CAs**)
  - Bob’s identity can be trusted, but he has no authority to sign others’ certificates
- Addressing scalability: Multiple trust anchors
  - There are ~150 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices
  - Each delegation step can restrict the scope of a certificate’s validity
  - Creating the certificates is an *offline* task: The certificate is created once in advance, and then served to users when requested

# Revocation

- What happens if a certificate authority messes up and issues a bad certificate?
  - Example: {“Bob’s public key is $PK_M$”}$_{SK_{CA}^{-1}}$
  - Example: Verisign (a certificate authority) accidentally issued a certificate saying that an average Internet user’s public key belonged to Microsoft

# Revocation: Expiration Dates

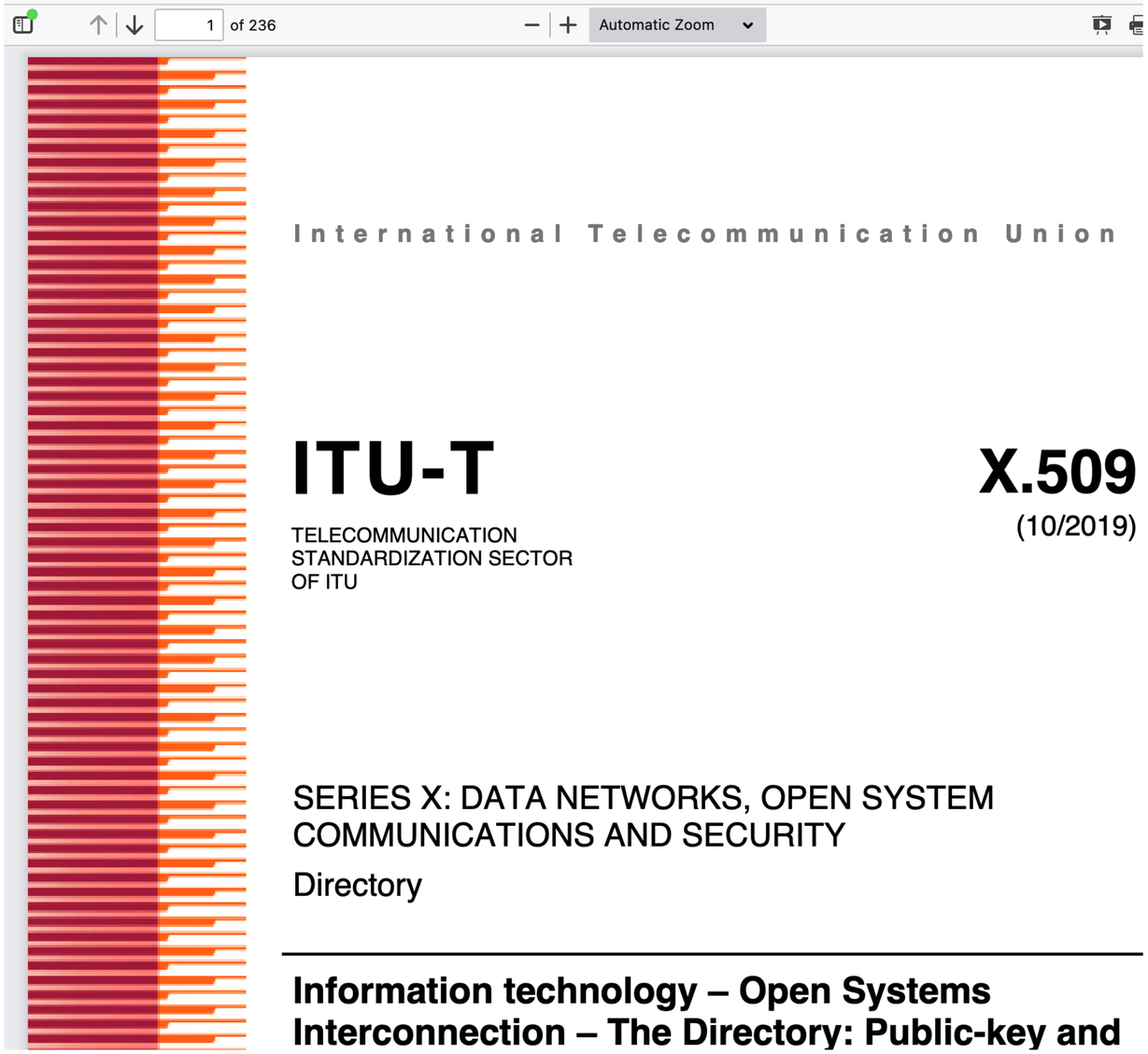
- Approach #1: Each certificate has an expiration date
  - When the certificate expires, request a new certificate from the certificate authority
  - The bad certificate will eventually become invalid once it expires
- Benefits
  - Mitigates damage: Eventually, the bad certificate will become harmless
- Drawbacks
  - Adds management burden: Everybody has to renew their certificates frequently
  - If someone forgets to renew a certificate, their website might stop working
- Tradeoff: How often should certificates be renewed?
  - Frequent renewal: More secure, less usable
  - Infrequent renewal: Less secure, more usable
- LetsEncrypt (a certificate authority) chose very frequent renewal
  - It turns out frequent renewal is more usable:
    It forces automated renewal instead of a once-every 3 year task that gets forgotten!

# Revocation: Announcing Revoked Certificates

- Approach #2: Periodically release a list of invalidated certificates
  - Users must periodically download a Certification Revocation List (CRL)
- How do we authenticate the list?
  - The certificate authority signs the list!
    - {“The certificate with serial number 0xdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
- Drawbacks
  - Lists can get large
    - Mitigated by shorter expiration dates (don’t have to list them once they expire)
  - Until a user downloads a list, they won’t know which certificates are revoked
- What happens if the certificate authority is unavailable?
  - Fail-safe default: Assume all certificates are invalid? Now we can’t trust anybody!
    - Possible attack: Attacker forces the CA to be unavailable (denial of service attack)
  - Use old list: Potentially dangerous if the old list is missing newly revoked certificates

# Certificates: Complexity

- Certificate protocols can get very complicated
  - Example: X.509 is incredibly complicated (a 236 page standard!) because it tried to do everything



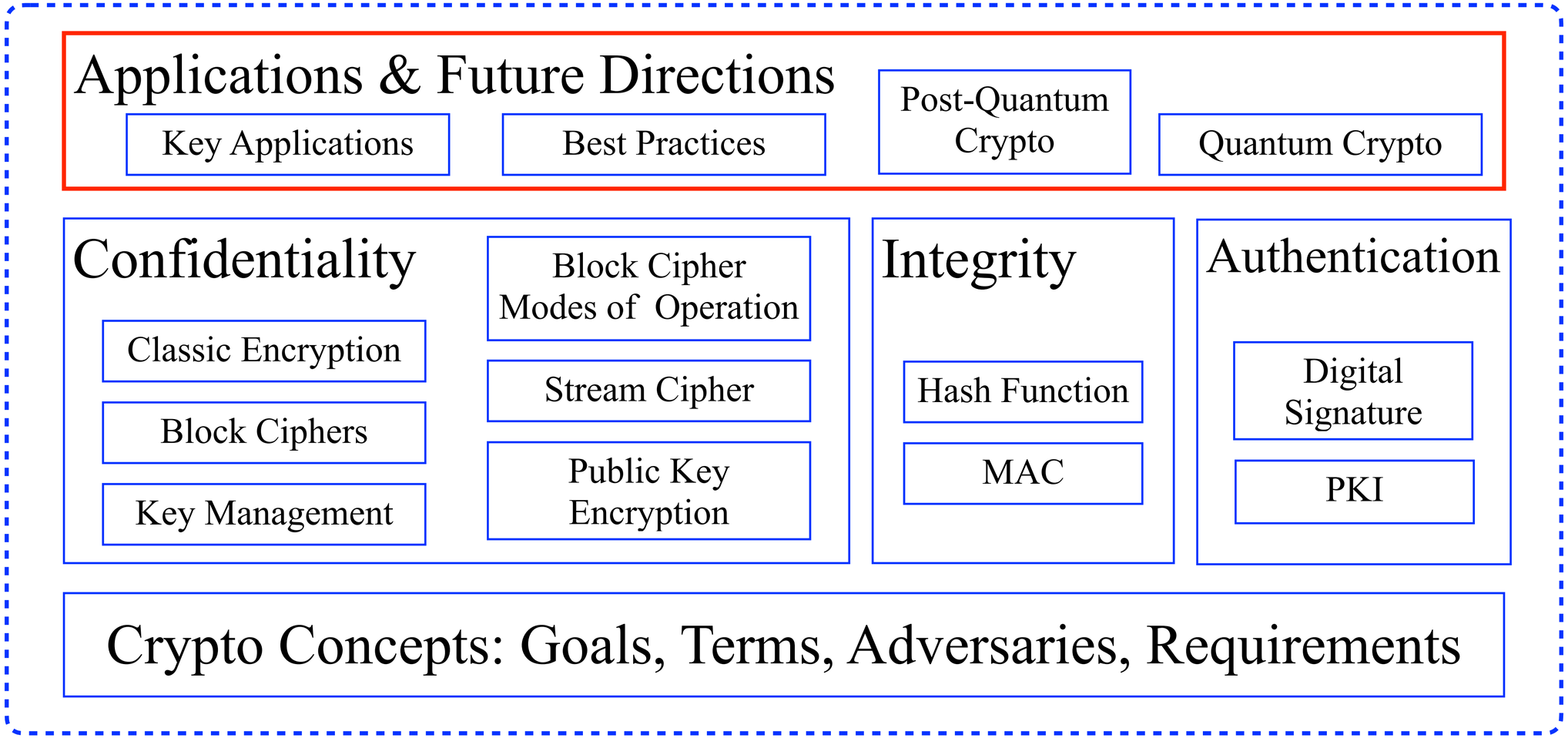
# Alternative: Web of Trust

- Modern public-key infrastructures are structured like trees
- Originally, public-key infrastructures looked like graphs instead
  - Everybody can issue certificates for anyone else
  - Example: Alice signs Bob's key. Bob signs Carol's key. If Dave trusts Alice, he trusts Bob and Carol.
  - Benefit: You know the trust anchor personally (e.g. because you met them in-person, or because you signed their key)
  - Problem: Graphs get far more complex than trees!
- OpenPGP (Pretty Good Privacy) originally used the web of trust model
  - Key-signing parties: meeting in-person to sign each other's public keys
  - It quickly proved to be a disaster
  - Instead, everyone just relies on MIT's central keyserver which is broken!
- **Takeaway**: Trust anchors make public-key infrastructures much simpler!

# Summary: Certificates

- Certificates: A signed attestation of identity
- Trusted directory: One server holds all the keys, and everyone has the TD's public key
  - Not scalable: Doesn't work for billions of keys
  - Single point of failure: If the TD is hacked or is down, cryptography is broken
- Certificate authorities: Delegated trust from a pool of multiple root CAs
  - Root CAs can sign certificates for intermediate CAs
  - Revocation: Certificates contain an expiration date
  - Revocation: CAs sign a list of revoked certificates

# The Roadmap of Cryptography



# Post-Quantum Crypto

- In Oct 2019, researchers from Google, claimed in a paper published in Nature that they have achieved quantum supremacy using a superconducting processor

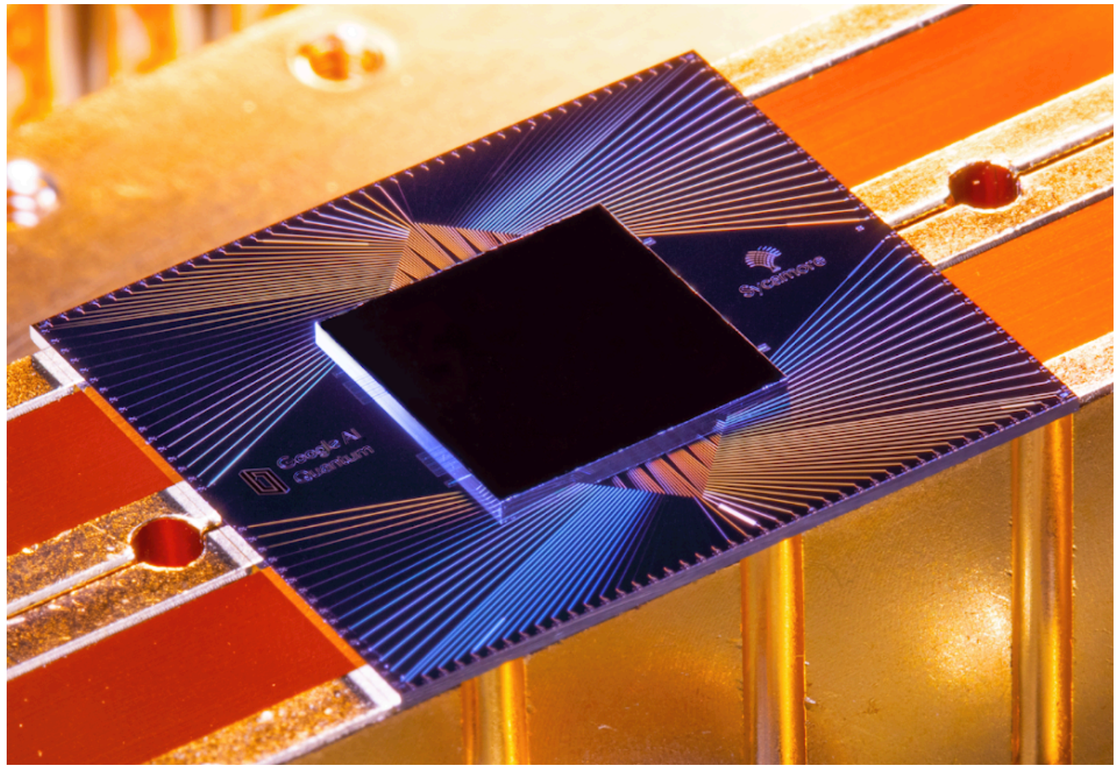
- Large-scale quantum computers, if built, would severely threaten RSA-based and discrete-log-based cryptographic algorithms, due to Shor's algorithm

- The Shor's algorithm allows polynomial-time algorithms for both IFP (integer factoring problem) and DLP (discrete logarithm problem)

# Post-Quantum Crypto

- On the other hand, Grover's algorithm, when running in a quantum computer, provides a quadratic speed-up for exhaustive key search (N -> $\sqrt{N}$)

- This motivates post-quantum crypto: the study of crypto algorithms and schemas that are secure against large-scaled quantum computers



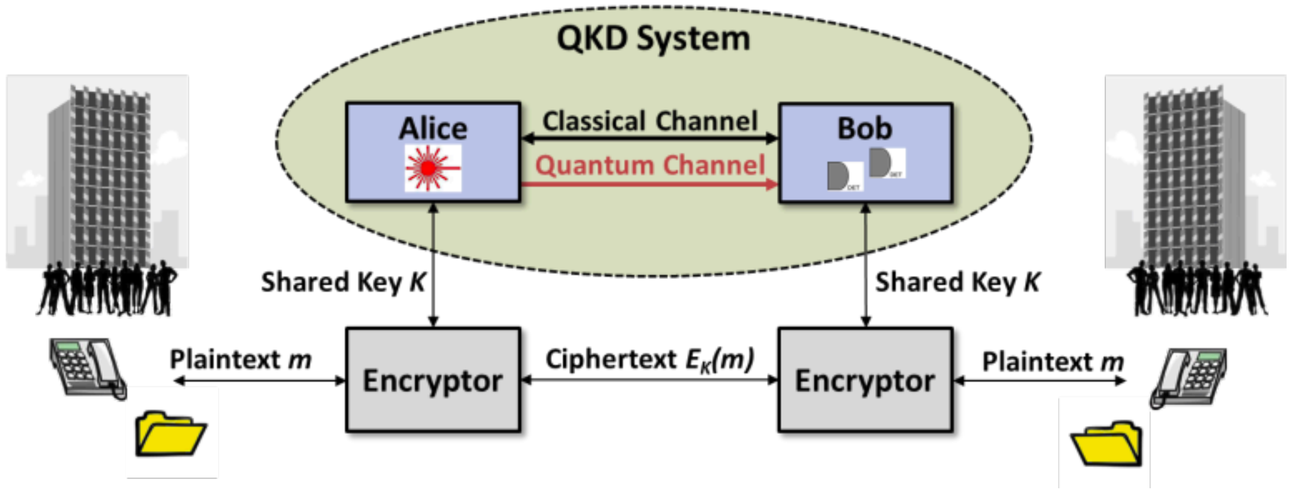
# Post-Quantum Crypto

- A list of post-quantum scheme proposals can be found as below.

https://csrc.nist.gov/projects/post-quantum-cryptography

# Quantum Crypto

- Different from post-quantum crypto, quantum crypto attempts to harness the power of quantum mechanics to build up cryptographic schemes
- One example is quantum key distribution (QKD)
- QKD allows the two communicating parties to detect eavesdropping ( the presence of any third party trying to gain knowledge of the key)



# Crypto Applications: TLS

- TLS 1.3 was released in 2018
- TLS is now used to protect over 95% of all HTTP traffic
- Key components
  - **TLS Handshake Protocol** uses (usually) asymmetric cryptography to establish session keys
  - TLS Record Protocol transfers encrypted data
  - TLS Alert Protocol can be used to transport management information and error alerts

# Crypto Applications: Secure Messaging

- Apple iMessage
  - end-to-end secure
  - Key distribution relies on trust of the central server
  - However, it doesn't fulfill  forward security (once a user's private decryption key was known, all messages intended for that user could be read)
  - However, **these observations are learned from reverse engineering,  and may no longer be accurate**

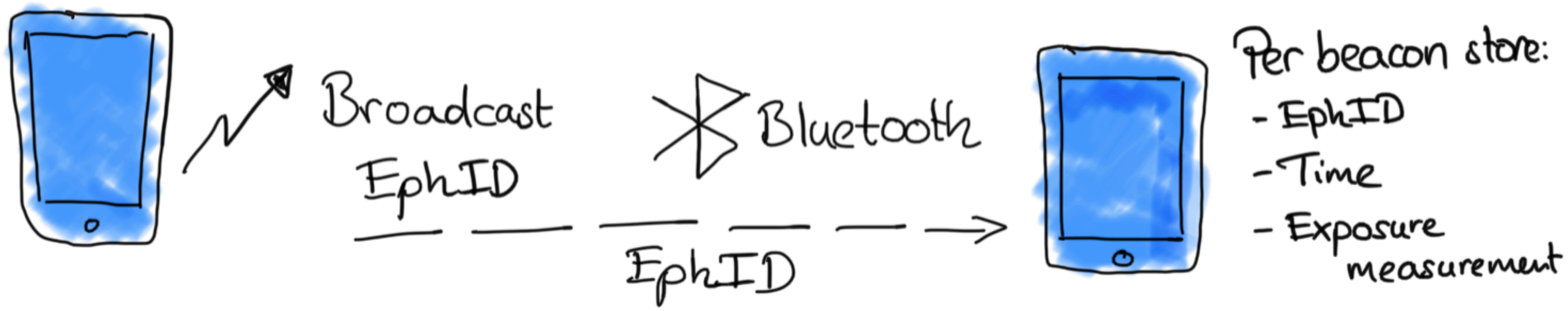
# Crypto Applications: Secure Messaging

- Signal
  - end-to-end secure
  - The double ratchet algorithm is built up to timely update key materials through Diffie-Hellman key exchange and KDF (Key Derivation Function)
  - It provides both forward security and backward security (post compromise security).
  - Primitives: AES with 256 bit keys for encryption, HMAC with SHA-256 for MAC,  Elliptic curve Diffie–Hellman (ECDH) with Curve25519

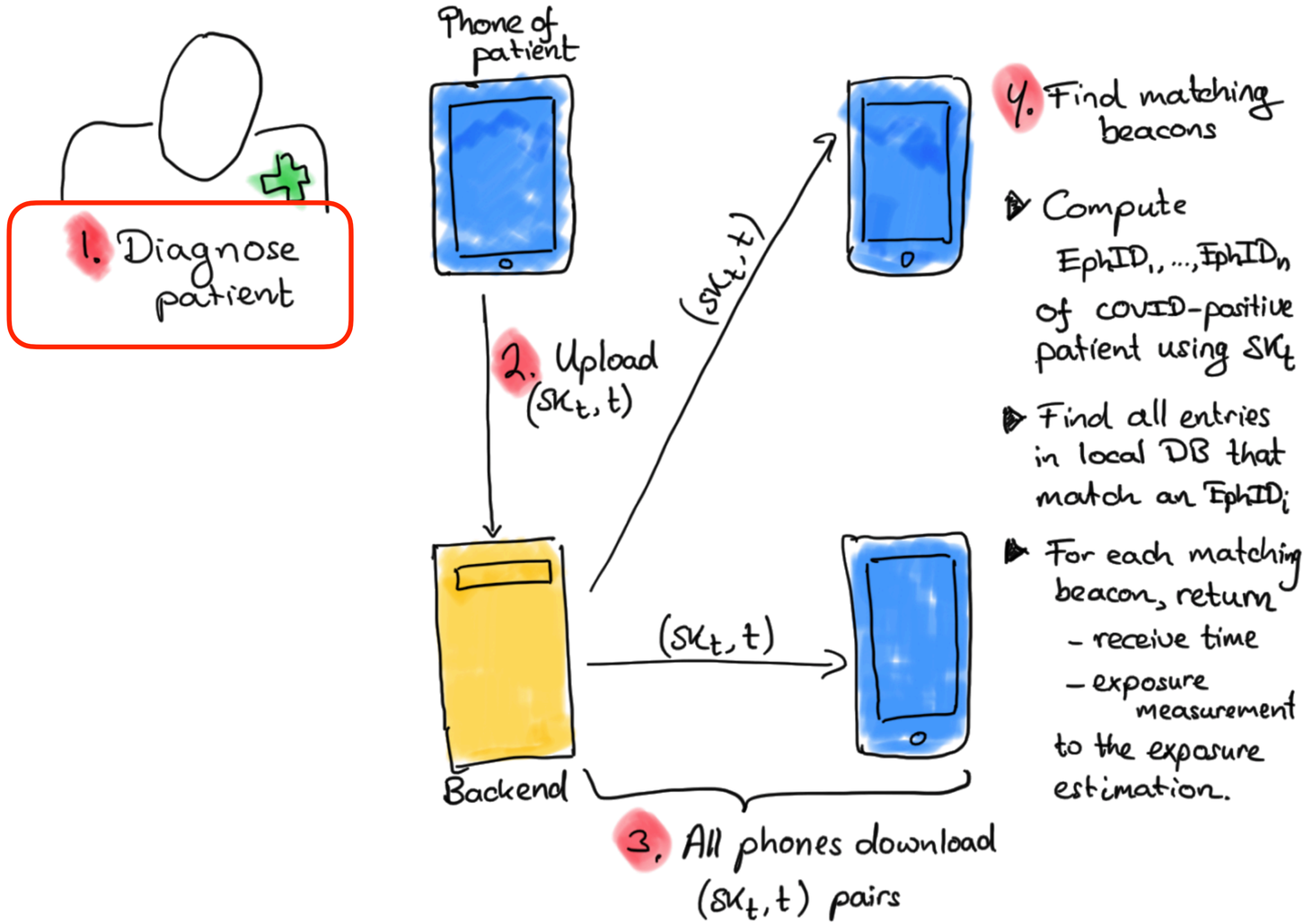
# Crypto Applications: Contact Tracing

- DP-3T, is a privacy-preserving contact tracing technology utilizing mobile phones and Bluetooth Low Energy beacons
- The Paper: Decentralized Privacy-Preserving Proximity Tracing
- *EphID* Generation: each day, smartphones rotate their secret day seed $\mathtt{SK}_t$ by computing

$$\mathtt{SK}_t = \mathtt{H}(\mathtt{SK}_{t-1}) \qquad \mathtt{EphID}_1||\ldots||\mathtt{EphID}_n = \mathtt{PRG}(\mathtt{PRF}(\mathtt{SKt}, \text{"broadcast key"}))$$



# Crypto Applications: Contact Tracing



# Best Practice of Crypto

- Key fallacies
  - Push the data through enough complicated steps and it must be secure
  - if an algorithm has a very large key space, then surely it must be secure
  - friendly cryptanalysis:  the inventor has tried to break the new algorithm themselves, so it must be secure
- Prefering standardized cryptographic solutions, to rolling new ones
- Making cryptography invisible and by default, e.g., Signal

# Q&A