CSE127, Computer Security

Cryptography

UC San Diego

Housekeeping

General course things to know

- PA4 due **tonight!**
  - Godspeed
- PA5 released **tomorrow**
  - Focuses on Cryptography… next unit in class
  - Last PA, due end of week 10
- Note, due to travel class is cancelled on **3/10**
- **Final exam location:** Mosaic Lecture Hall 113 (has 250 seats so we don't need to be so cramped!)

Previously on CSE 127…

Recap

- We've been all over the map!
  - App Sec, Systems, Web, Networks…
- We've learned about important security principles
  - Confidentiality, Integrity, Availability, Trust, Privacy….

Today's lecture — Intro to Cryptography

Learning Objectives

- Learn about cryptography — what it is, how we use it, and the guarantees that cryptography gives us
- Discuss message *integrity* and message *confidentiality* and how we are able to enable both via *symmetric cryptography*
- Learn about hashes, message authentication codes, encryption… get a little bit into the math of it all (but not too much, don't worry)

Cryptography Basics

A simple example

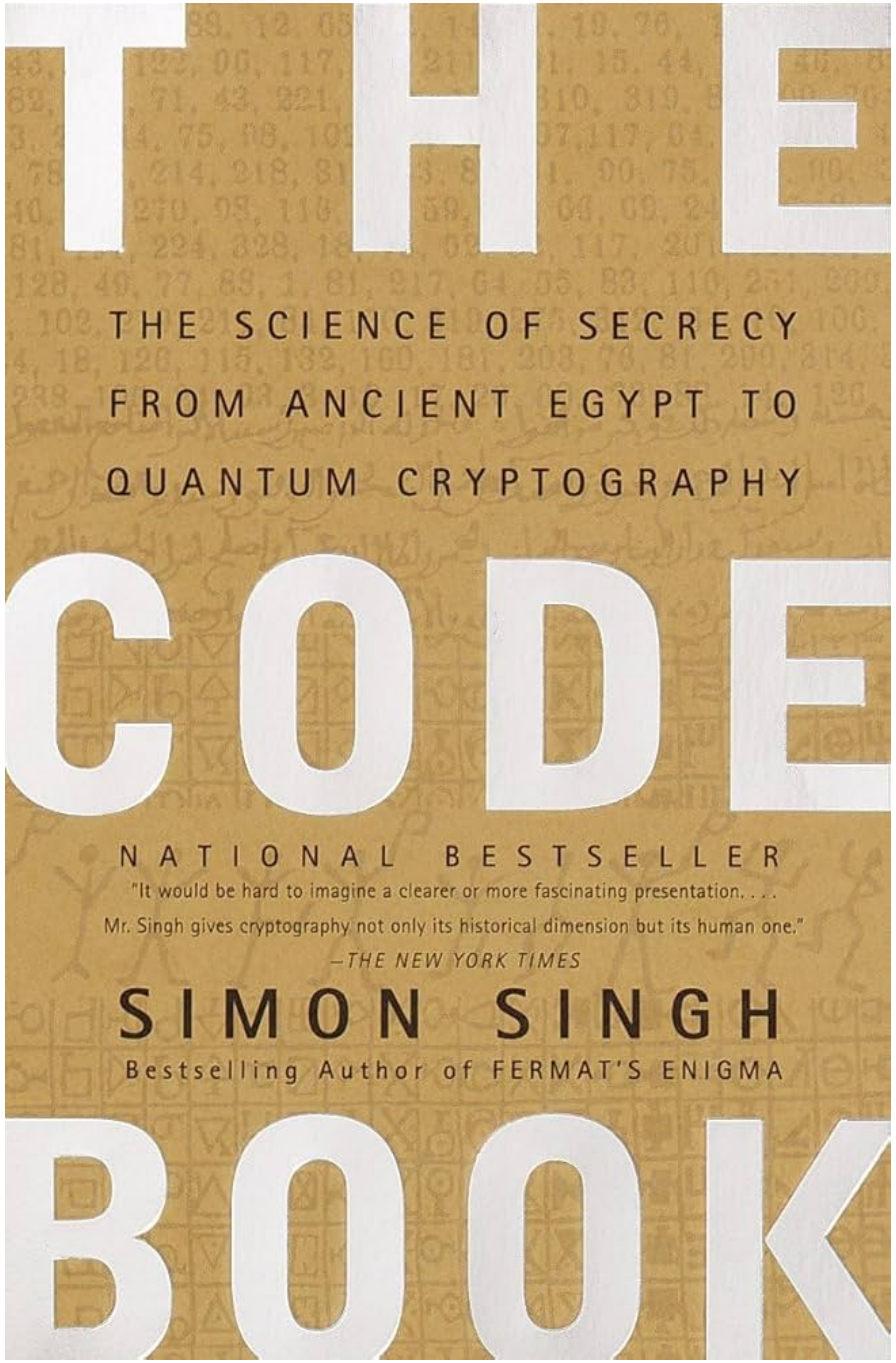
- Alice wants to pass a note to Bob about whether Bob will work with her on PA5
- …but in order to pass the message to Bob, she needs to go through Eve
- Alice does not want Eve to know she's asking Bob, because Eve was her PA4 teammate (and Eve sucks)
- …**what can Alice do?**

Cryptography

- **Cryptography** provides mechanisms for enforcing confidentiality and integrity across time and space controlled by an adversary
- *Very* broad subject
  - Core mathematics, groups, rings, abstract algebra
  - Cryptography primitives (hashes, signatures, encryption algorithms)
  - Applications (SSH, SMTPS, HTTPS, etc.)
  - Post-quantum cryptography
  - …and more!
- Take CSE107 / 207 if you're interested in *actually* learning it deeply, here we're just giving you a sampler

Motivations of Cryptography

- Cryptography (and security, broadly) has its origins in war / defense
    - Julius Caesar used the "Caesar Cipher" to encrypt wartime messages and clandestine communication
- Two parties want to communicate securely:
    - Confidentiality: No one else can read messages
    - Integrity: Messages cannot be modified
    - Authenticity: Parties cannot be impersonated



Recall: Attacker models

- Passive eavesdropper (Eve)
  - Can read messages only
- Person-in-the-middle attacker (Mallory)
  - Can read, create, modify, block messages

Back to our simple example!

- Let's say Alice and Bob *pre-agree* on a secret code:
  - "Eagle": "yes"
  - "Snake": "no"
- Alice sends the message "eagle"
  - <u>What does Eve learn from reading this?</u>

Back to our simple example!

- Let’s say Alice and Bob *pre-agree* on a secret code:
  - “Eagle”: “yes”
  - “Snake”: “no”
- Alice sends the message “eagle”
  - What does Eve learn from reading this?
- Eve learns basically nothing, except…
  - Now she knows Alice is sending a message “eagle,” and if she’s already suspicious of Alice for being a traitor, maybe she can mess with their channel…

Back to our simple example!

- Let’s say Alice has sent another message to Bob, saying “snake”
  - <u>Does their communication channel have confidentiality?</u>

Back to our simple example!

- Let's say Alice has sent another message to Bob, saying "snake"
  - Does their communication channel have confidentiality?
    - **Yes**, Eve doesn't know what "snake" means!
  - Does their communication channel have integrity?

Back to our simple example!

- Let's say Alice has sent another message to Bob, saying "snake"
  - Does their communication channel have confidentiality?
    - **Yes**, Eve doesn't know what "snake" means!
  - Does their communication channel have integrity?
    - **No!** Eve can replace the message with whatever (maybe when Alice and Bob are not looking), like "eagle"
  - Does their communication channel have authenticity?

Back to our simple example!

- Let’s say Alice has sent another message to Bob, saying “snake”
  - Does their communication channel have confidentiality?
    - **Yes**, Eve doesn’t know what “snake” means!
  - Does their communication channel have integrity?
    - **No!** Eve can replace the message with whatever (maybe when Alice and Bob are not looking), like “eagle”
  - Does their communication channel have authenticity?
    - **No!** Bob doesn’t know where the message came from, he only knows he’s expecting a message from Alice by way of Eve

Different properties require different crypto

- Know your threat model!
- Know whether you need to protect ***confidentiality, integrity,*** *or* both.

Different properties require different crypto

- Know your threat model!
- Know whether you need to protect ***confidentiality, integrity,*** *or* both.

Confidentiality and Integrity are protected by different cryptographic mechanisms! Having one does not imply the other!!!!

Different attackers require different crypto

- Know your threat model!
- Know whether you need protection against a ***passive*** or ***active*** adversary.

Different attackers require different crypto

- Know your threat model!
- Know whether you need protection against a ***passive*** or ***active*** adversary.

Systems that are secure against the former may not be secure against the latter.

Cryptographic classifications

- **Symmetric** cryptography
  - Alice and Bob share a ***secret key*** that they use to secure their communications
  - Secret keys are random bit-strings
- **Asymmetric** cryptography
  - Each subject has two keys: public and private
  - ***Public keys*** can be used by anyone for “unprivileged” operations — encrypt message for intended receivers
  - ***Private keys*** are secret and used for “privileged” operations — decrypt message

Cryptographic primitives

- **Message authentication codes (symmetric) and Digital signatures (asymmetric):** Provide integrity and authenticity, no confidentiality
  - Formally: adversary can't generate a valid MAC or signature for a new message without knowing the secret key
- **Encryption:** provides confidentiality, without integrity protection
  - Formally: adversary can't distinguish which of the two plaintexts were encrypted without knowing the secret key

Symmetric Key Cryptography

Goal: Message Integrity

- Alice wants to send message ***m*** to Bob, but…
  - We don't trust the messenger!

- As defender, we want to be sure that **what Bob receives** is identical to **what Alice sent**

Goal: Message Integrity

- **Threat Model**
  - Mallory can see, modify, and even forge messages
  - Mallory wants to trick Bob into accepting a message Alice didn't send!

One idea…

- Alice computes ***v = f(m)***
- E.g., ***m =*** "Attack at dawn," *f(m) = 5892294875….*
- Bob verifies that **v' = f(m')**, and accepts the message if and only if this is true.
  - Does this offer message integrity? Why or why not?

Selecting a function *f*

- Bob accepts the message iff **v′ = *f(m′)***
- We want ***f*** to be easily computable by Alice and Bob, but **not** computable by Mallory. **How?**

Hash functions

- A ***cryptographic hash function*** maps arbitrary length input into a fixed-size string.
  - <u>Do hash functions always produce unique output?</u>

Hash functions

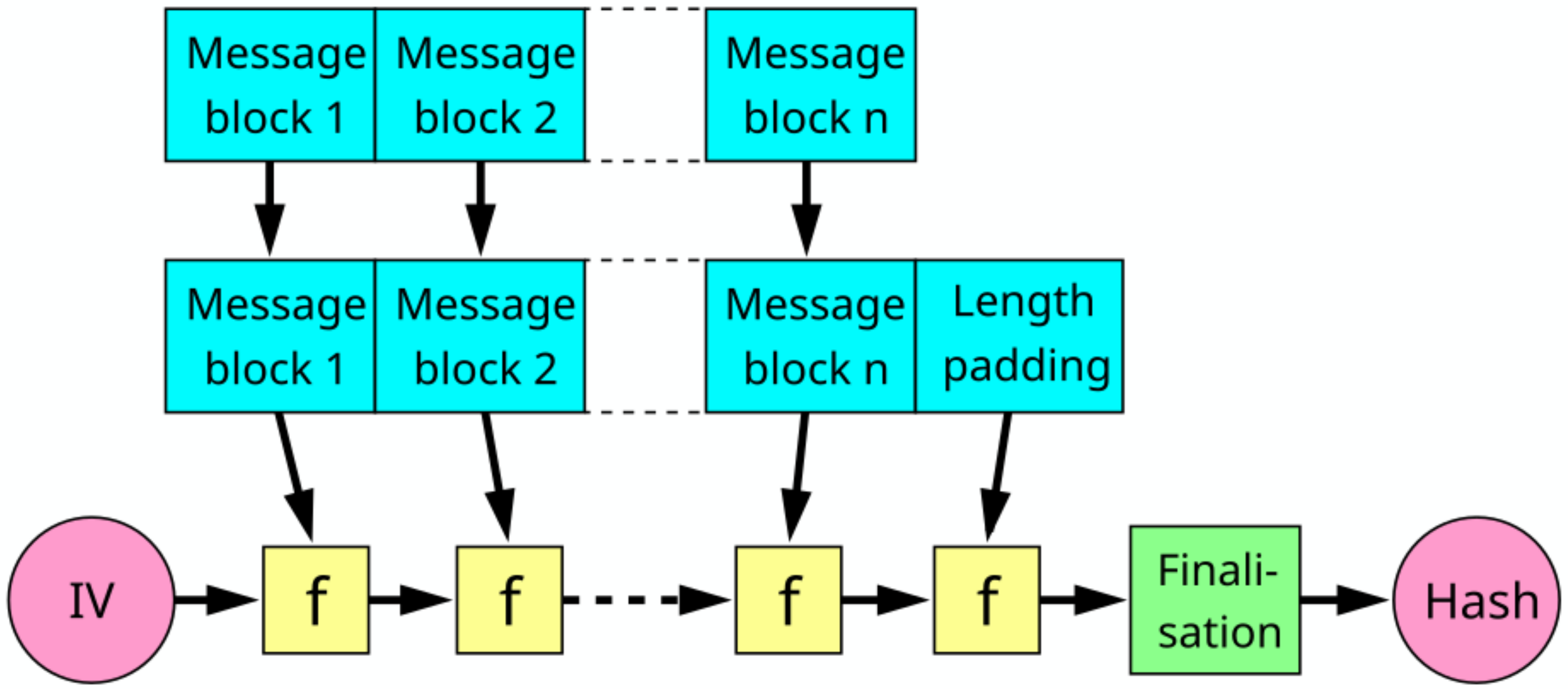
- A ***cryptographic hash function*** maps arbitrary length input into a fixed-size string.
  - <u>Do hash functions always produce unique output?</u>
- Two desirable properties of hash functions
  - ***Pre-image resistance***
    - Given a specific hash function output, it is impractical to find an input (pre-image) that generates the same output
  - ***Collision resistance***
    - It is impractical to find any two inputs that hash to the same output.

Hash functions

- Some examples include….
  - MD5 — broken in 2004, very easy to find collisions that can do bad things, and can be exploited to attack real systems! You'll break this in PA5.
  - SHA1 — broken in 2017, decently hard to find collisions but not impossible
  - SHA2 — Designed by NSA, output 224, 256, 384, 512 bits, **not broken**
  - SHA3 — Result of NIST SHA-3 contest, produces many different bit-size outputs, recommended for **all new applications**

How do hash functions work?

- MD5, SHA-1, SHA-2 are all built using the **Merkle-Damgård construction**
  - Each hash function has a unique ***compression function*** f, which takes in two fixed sized inputs and produces one fixed size output.

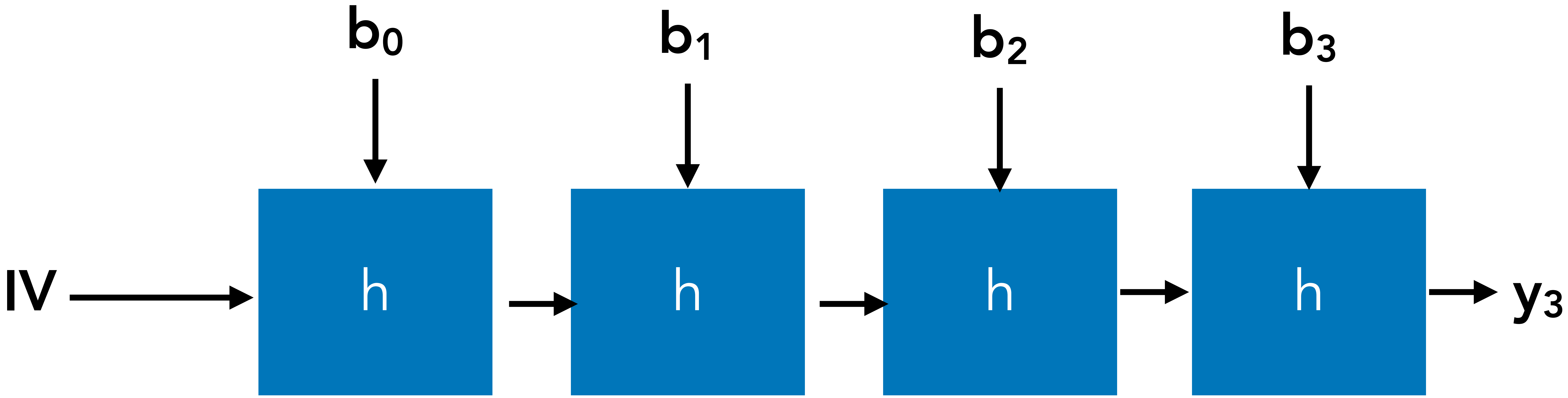


Constructing SHA-256

- Input: Arbitrary length data
  Output: 256-bit digest
- Built with compression function ***h*** – operates on 512-bit blocks with a 256-bit output
- Basic idea: Pad input ***m*** to multiple of 512 bits (with standard padding), split m into 512-bit blocks $\mathbf{b_0, b_1, b_2 \ldots b_{n-1}}$
- $\mathbf{y_0 =}$ constant initialization vector…. $\mathbf{y_1 = h(y_0, b_0), \ldots. yi = h(y_{i-1}, b_{i-1})}$
- Return $\mathbf{y_n}$

Constructing SHA-256

- Basic idea: Pad input ***m*** to multiple of 512 bits (with standard padding), split m into 512-bit blocks $\mathbf{b_0, b_1, b_2 ... b_{n-1}}$
- $\mathbf{y_0}$ = constant initialization vector…. $\mathbf{y_1 = h(y_0, b_0), ... yi = h(y_{i-1}, b_{i-1})}$
- Return $\mathbf{y_n}$



Merkle-Damgård is vulnerable to length extension attacks!

- You will exploit this in PA5!
  - Basic idea…. Given H(x) and len(x), you can construct H(x||y) *without every learning* x (here,|| means concatenation to cryptographers)
  - **Why?** You can load the state of the function in the previous block and infer the padding from the length (allowing any arbitrary extension)
  - Can lead to some bad outcomes… e.g.
    - Using hash(secret || message) as a verification mechanism (was very popular in the early 2000s)

Using a hash as *f*

- Let's say Alice uses SHA-3 as her integrity function and send the message over to Bob.
    - ***v = SHA-3(m)***
- Does this provide Bob with an integrity guarantee?

Using a hash as *f*

- Let's say Alice uses SHA-3 as her integrity function and send the message over to Bob.
  - ***v = SHA-3(m)***
- Does this provide Bob with an integrity guarantee?
  - No! Because Mallory can just as easily construct SHA-3(m)…. hm……

Enter: Message Authentication Codes

- Goal: Validate message integrity and authenticity based on a ***shared secret.***
  - How can Bob know that the message is really from Alice and not been modified by Mallory?
- MAC: Message Authentication Code
  - Function of message **and a secret key**
  - Impractical to forge without knowing the key
    - i.e., to come up with a valid MAC for a new message

MACs in action

- Alice sends the following: **m, a = $\mathbf{MAC_k(m)}$**
- Bob uses his copy of the secret key *k* to independently compute **a'** on **m** and compare to the one he received
- Mallory <u>cannot</u> compute MAC because she doesn't know the secret key!
    - Note, **no confidentiality guarantees.**

State of the art: HMAC

- HMAC: Hash-based Message Authentication Code
  - Use a hash in combination with a MAC to provide both *authenticity* and *integrity*
  - Hash provides message integrity, key provides authenticity!

$$\mathrm{HMAC}(K, m) = \mathrm{H}\Big(\big(K' \oplus opad\big) \parallel \mathrm{H}\big(\,(K' \oplus ipad) \parallel m\big)\Big)$$

$$K' = \begin{cases} \mathrm{H}(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

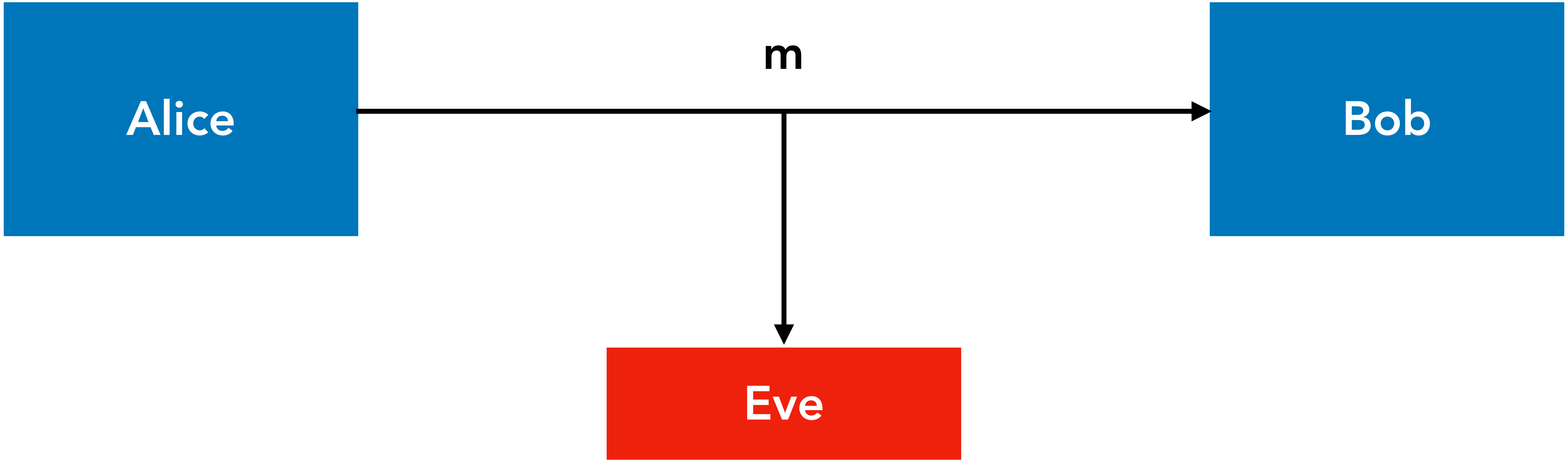
https://en.wikipedia.org/wiki/HMAC

…which brings us to Kerckhoff's Principle

- ***Kerckhoff's Principle:*** A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.
    - Deepak's version: Security by obscurity is *not* security!
- Claude Shannon's maxim: "the enemy knows the system"
    - i.e., "one ought to design systems under the assumption that the enemy will immediately gain full familiarity with them"
- Assume all details of the algorithm are public
    - **Only the key is secret!**

Goal: Message Confidentiality

- Alice wants to send a message to Bob, but we want to keep the contents of the message *secret* from a passive eavesdropper Eve



Enter: Encryption

- Encryption: converting data into something secret with some code
  - In general — input is a binary string of arbitrary length
  - We usually use a **cipher –** a mechanical algorithm for transforming plaintext to/from ciphertext
- Plaintext (m): unencrypted message to be communicated
  - Assume this is a binary string
- Ciphertext (c): encrypted version of the message
  - Also a binary string, but may not be the same length as the plaintext

Idea: One-time pad

- We can achieve **perfect secrecy** if we XOR plaintext with a random stream of bits known only to Alice and Bob. **Why?**

$$c = m \oplus r$$

Plaintext (a binary string)

Random binary string of the same length as plaintext

Idea: One-time pad

- Well, for any given ciphertext, every plaintext is **equally probable…** but…
  - You can never use the same key twice
  - Requires Alice and Bob to share a lot of pre-arranged secrets all of the lengths of message they're trying to share
  - Really hard to make this happen in practice.

$$c = m \oplus r$$

Plaintext (a binary string)

Random binary string of the same length as plaintext

Early ideas: Caesar Cipher!

- First recorded use: Julius Caesar (100 – 44 BCE)
  - Replaces each plaintext letter with the letter a fixed number of places down the alphabet
  - Encryption: $\mathbf{c_i = (p_i + k) \bmod 26}$
  - Decryption: $\mathbf{p_i = (c_i - k) \bmod 26}$
- P ht h wyvmlzzvy —> I am a professor (k = 7)

More complicated: Vigenére Cipher

- Encrypt alphabetic text where each **letter** of plaintext is encoded with a Caesar cipher depending on a *key*

```
ATTACKATDAWN
DUMDUMDUMDUM
____________
DNFDWWDNPDQZ
```

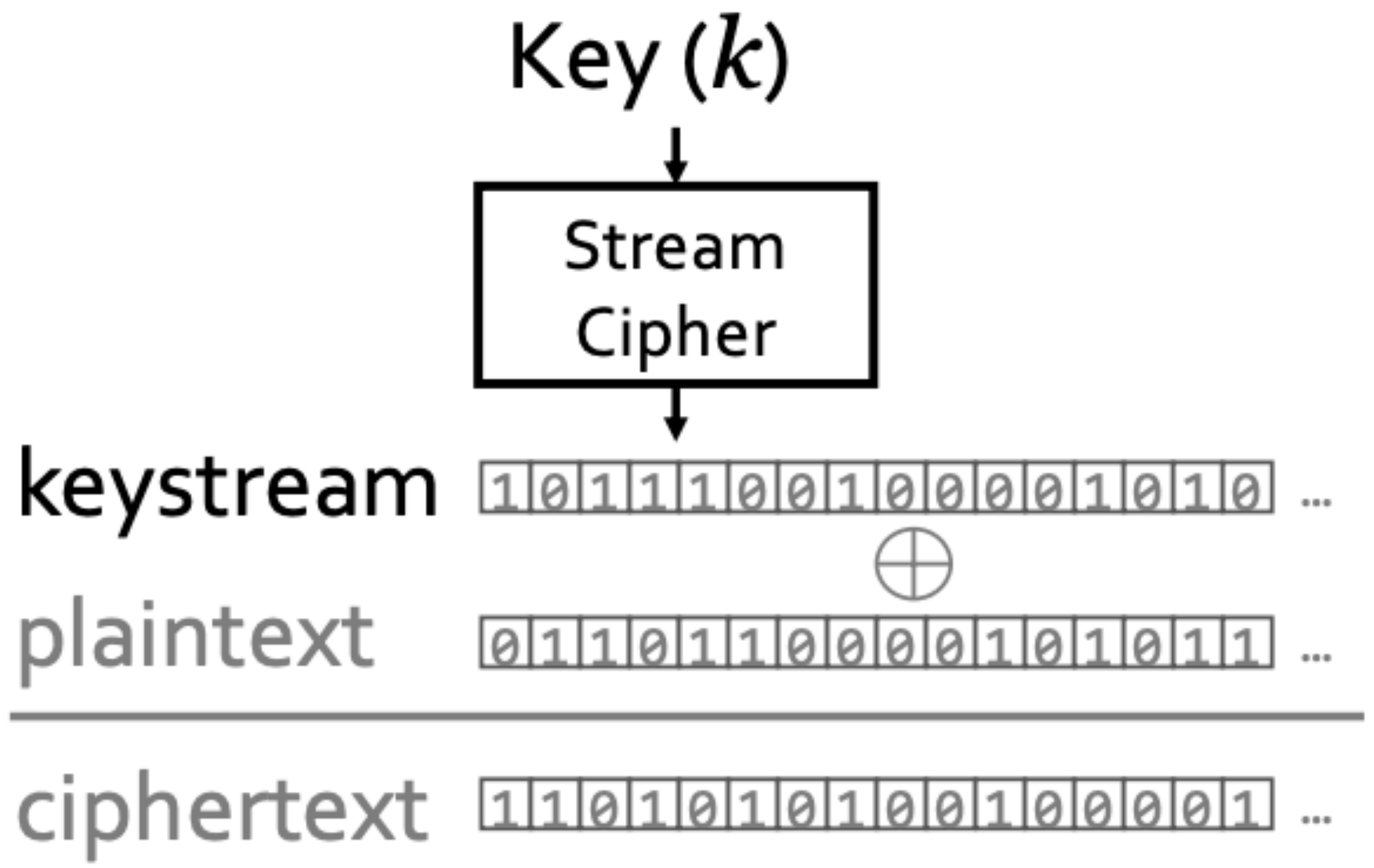
- Typically broken using something called the Kasiski method, or a form of *cryptanalysis*…. you'll do this in PA5!

Nowadays…

- For modern encryption, we use keyed functions (with non-easily guessable keys….)
- Two classes of symmetric encryption:
  - **Stream cipher:** generate a pseudorandom string of bits as long as the plaintext and XOR w/ the plaintext
    - Pseudorandom: hard to tell apart from random; cryptographers have a lot of formalism around this
  - **Block cipher:** Encrypt/decrypt fixed-size blocks of bits
    - Need a way to encrypt longer or shorter messages

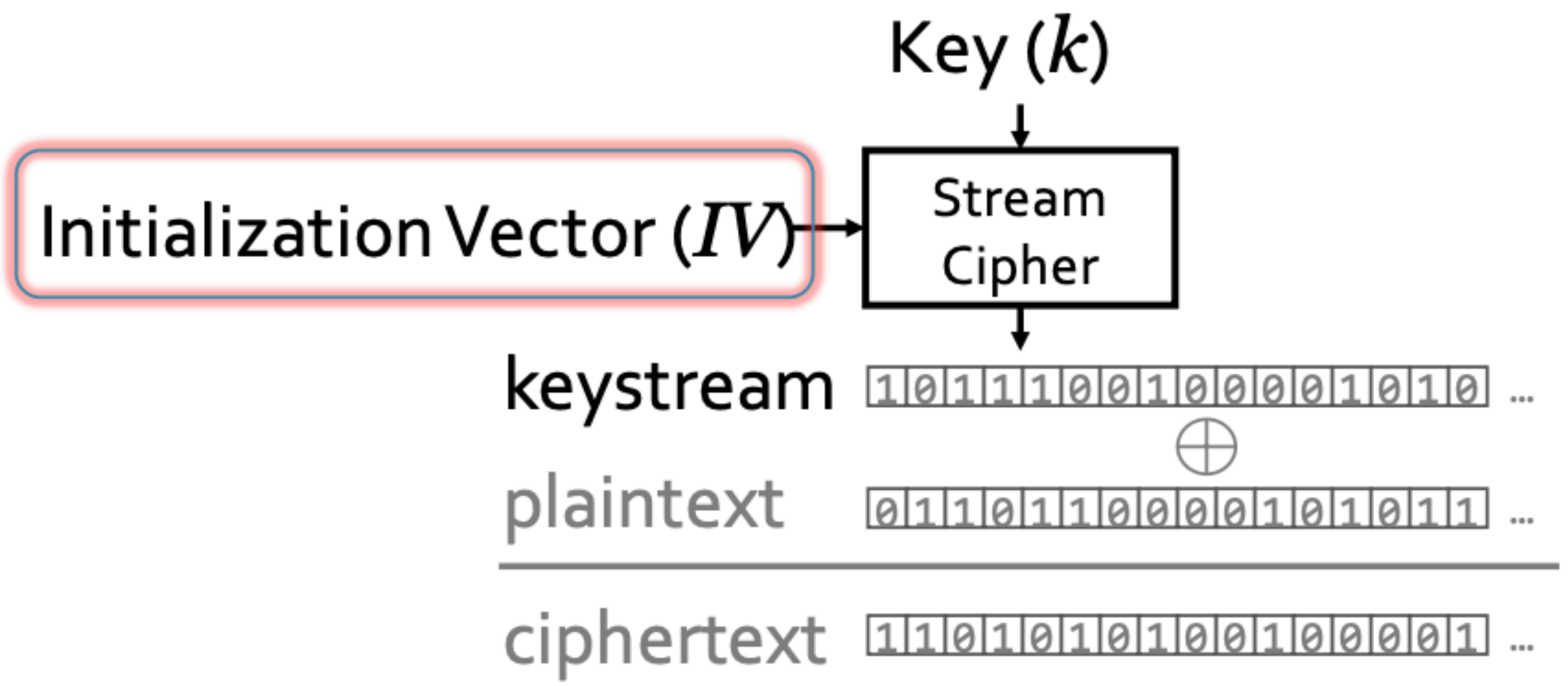
Stream Ciphers

- Produce a pseudorandom ***keystream***
  - Each key results in a unique, pseudorandom keystream
- To encrypt, keystream is XORed with plaintext
- To decrypt, keystream is XORed with ciphertext



Stream Ciphers

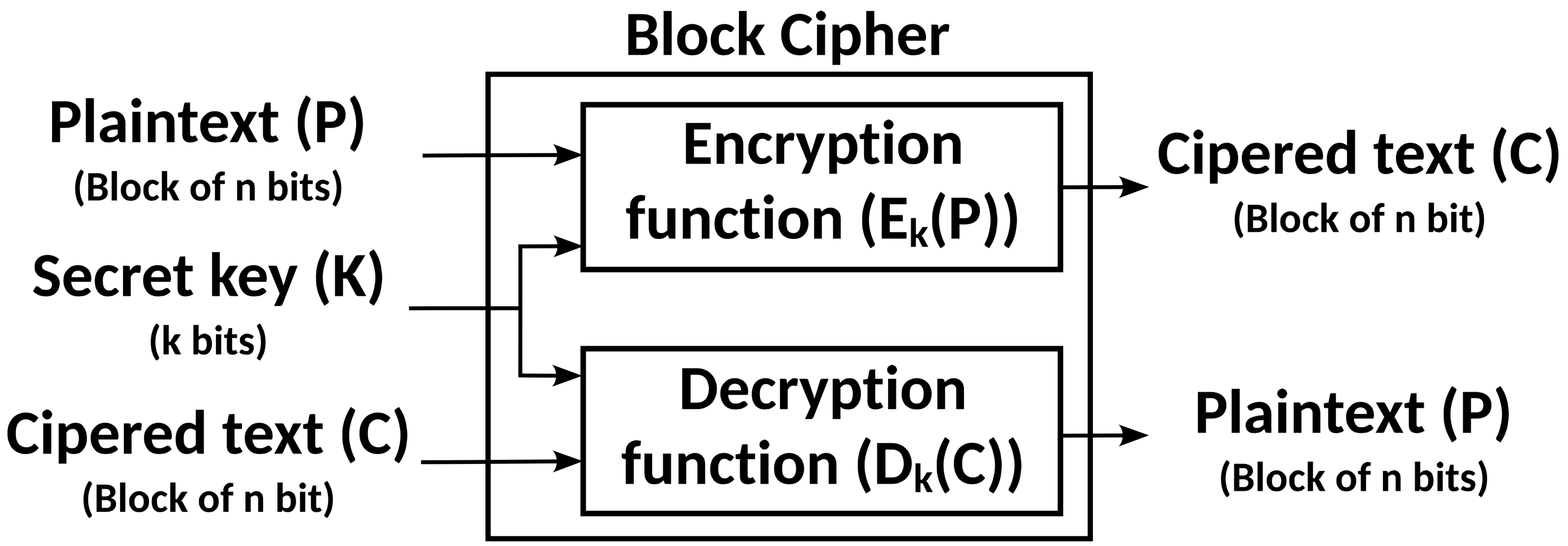
- Note: Insecure if key used more than once!
  - Need mechanism to either…
    - Generate a different one-time key from a master key, or…. a random **initialization vector** on each use (just like a compression function!)



Block Ciphers

- Block ciphers operate on fixed-size blocks
  - Common sizes: 64 and 128-bits
- A block cipher is typically a combination of a **permutation**
  - Each input is mapped to exactly one output
- And **substitution**
  - Some codewords are mapped to other codewords
- Typically in *multiple rounds*
- Examples: DES, AES, Blowfish…

Block Ciphers



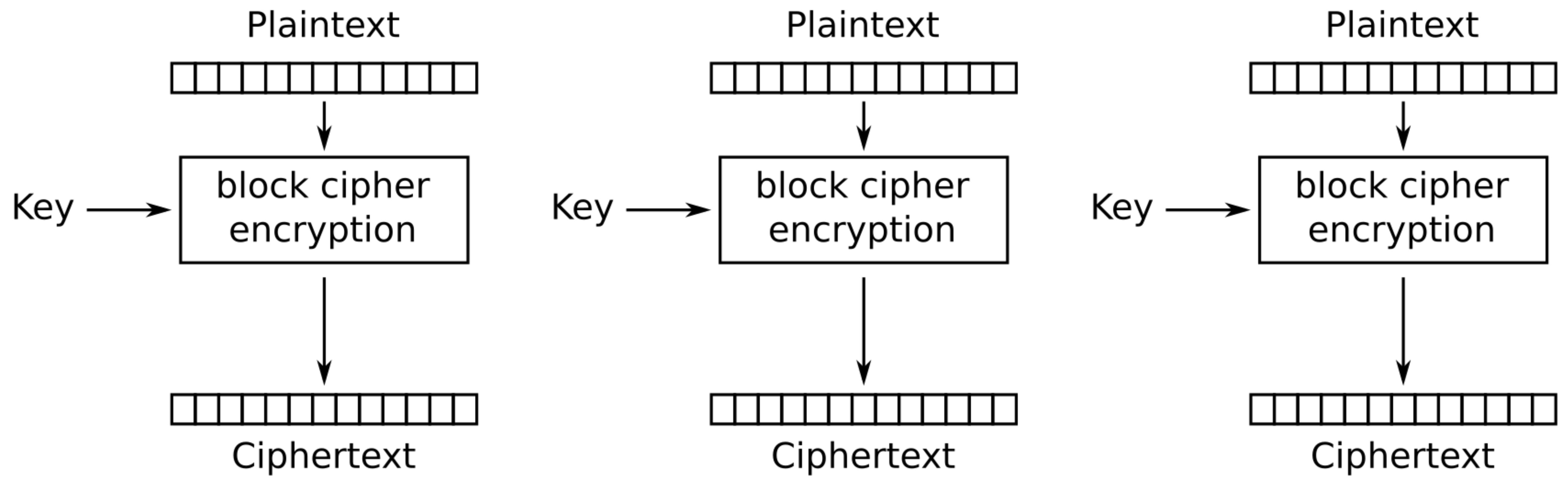
The inner workings are complicated; we'll ignore them (take 107 if you want)

Block Ciphers

- Okay, so… how do we use block ciphers on messages longer than the block size?
  - Pad plaintext to full block size!
  - Must be able to *unambiguously distinguish padding from plaintext*
  - **Don’t make up your own padding scheme!**
- How to encrypt a message longer than a block?
  - “Chain” individual blocks together
  - Methods of chaining are known as ***modes of operation***

Electronic Code Book (ECB) Mode

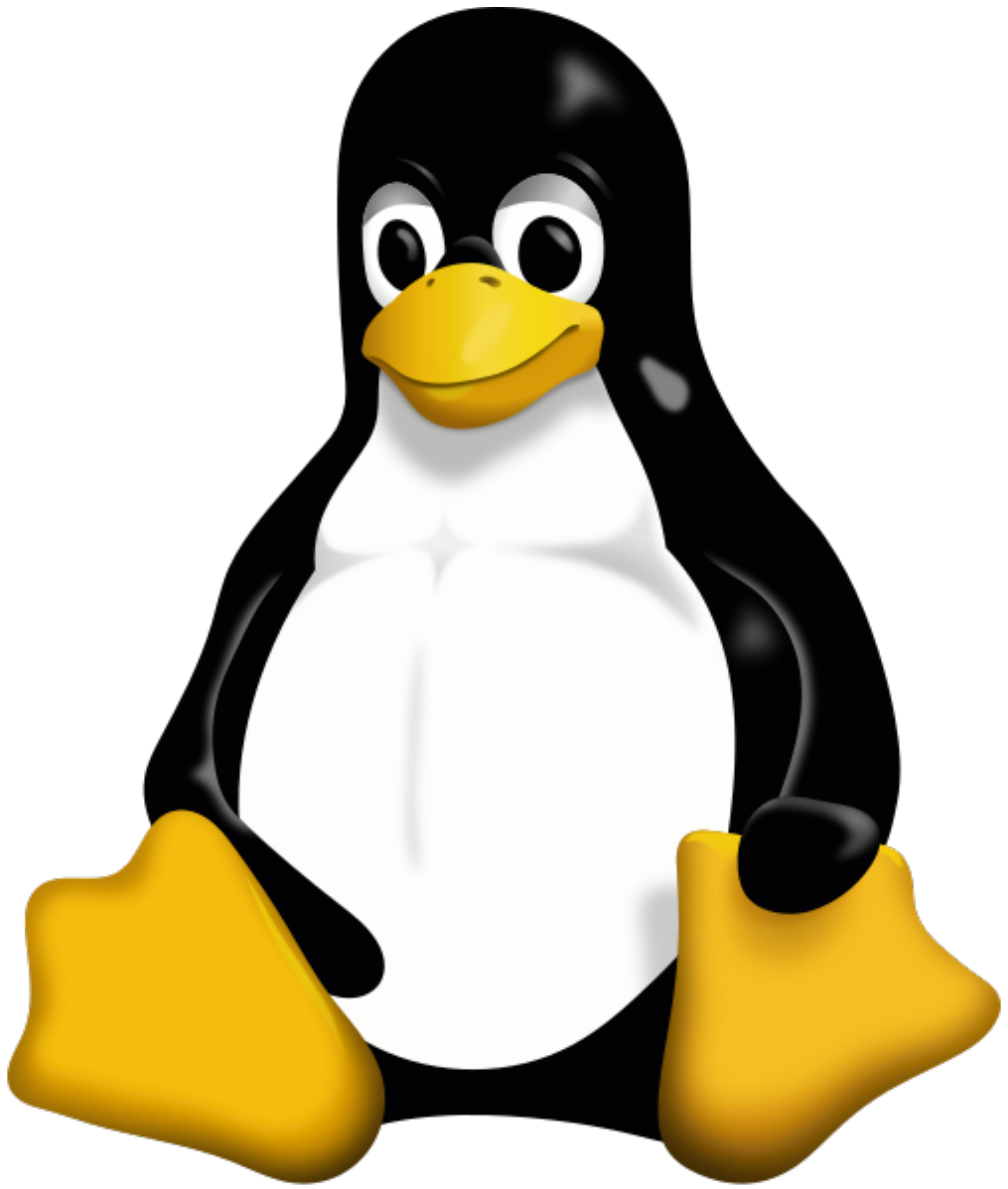
- Naive mode of operation: encrypt each block separately
  - Very fast (easy to parallelize)
- **DO NOT USE WITHOUT A GOOD REASON!**



Electronic Codebook (ECB) mode encryption

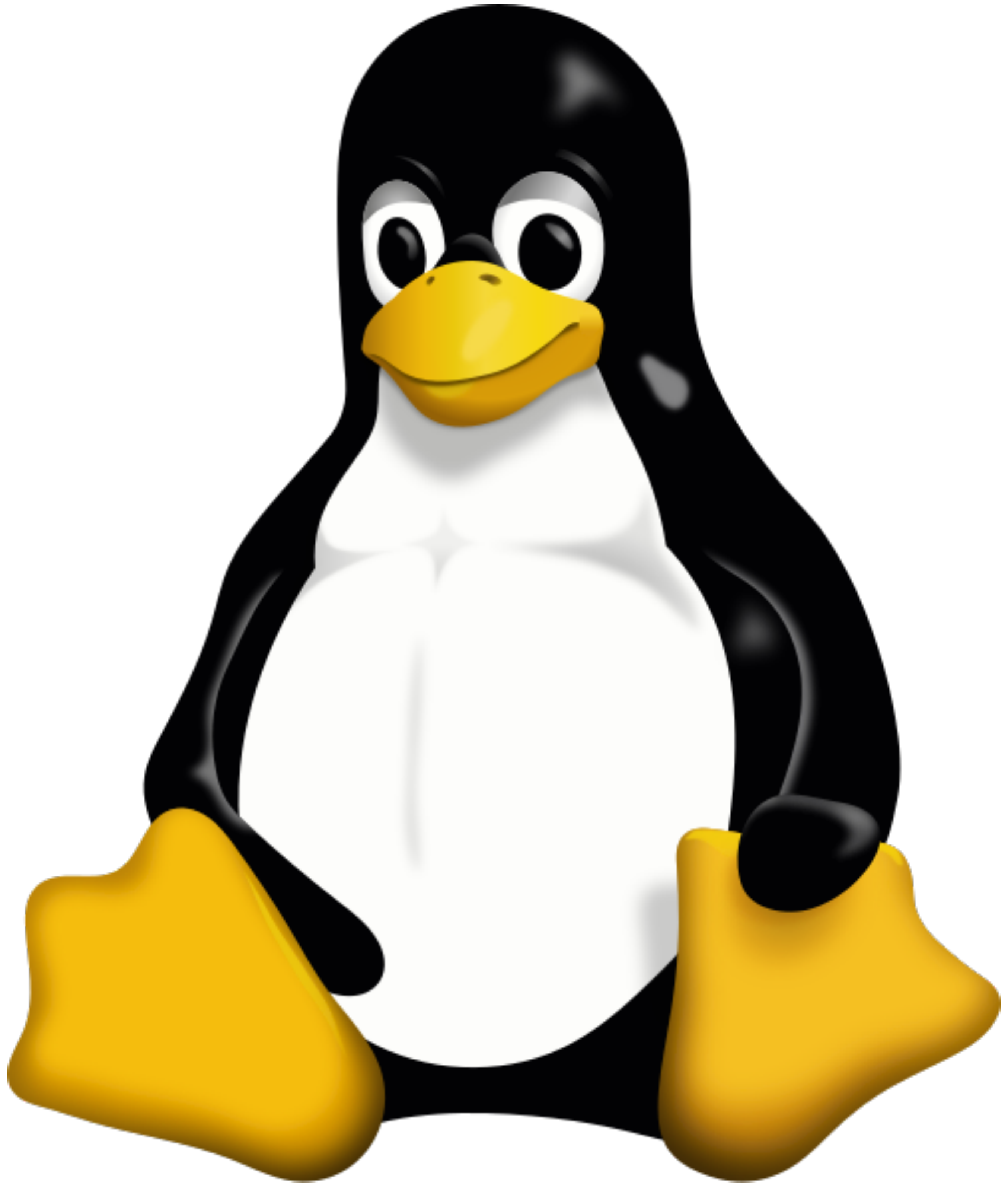
Electronic Code Book (ECB) Mode

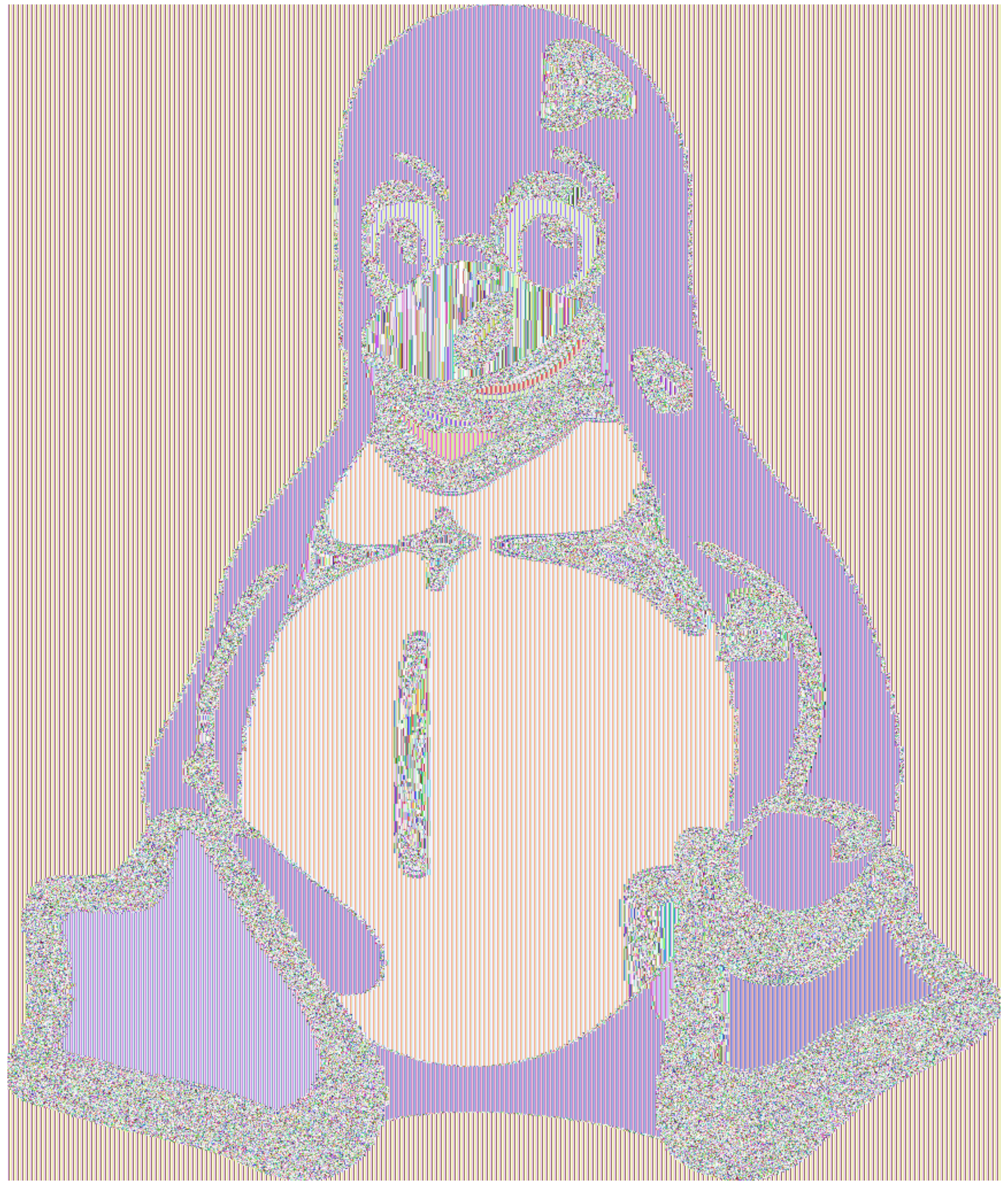
- What happens if we encrypt Tux in ECB mode?



Electronic Code Book (ECB) Mode

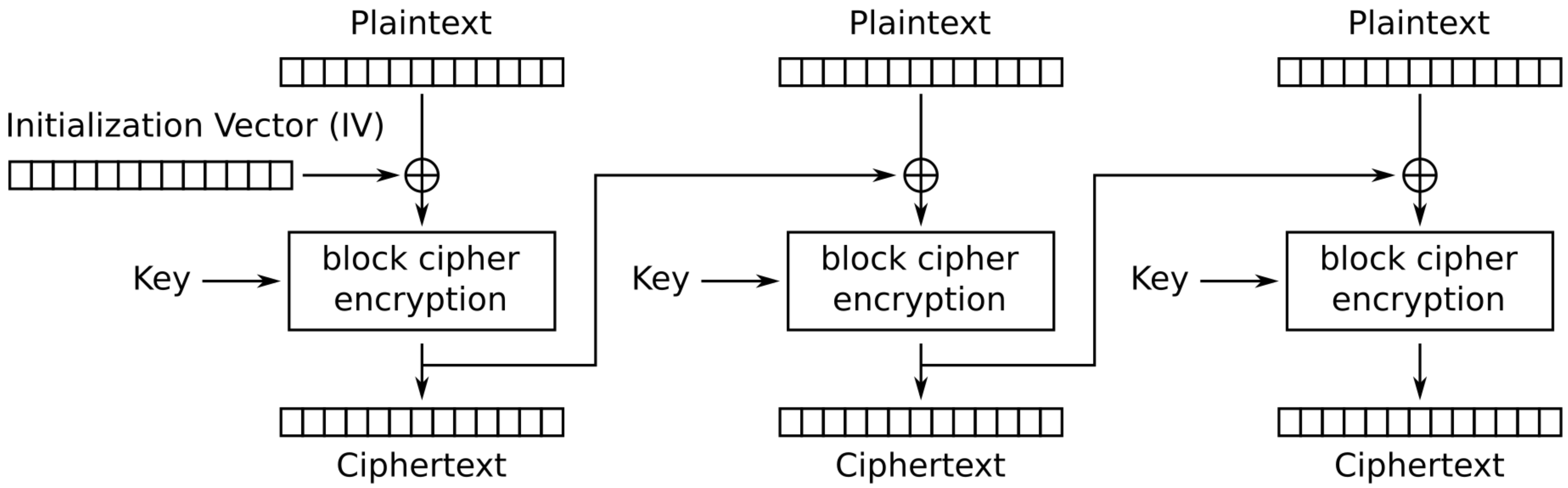
- What happens if we encrypt Tux in ECB mode? **All the same block values map to the same thing, leaking a lot of information.**





Cipher Block Chaining (CBC) Mode

- XOR ciphertext block into the next plaintext
- Use a random IV
  - Subtle attack possible if attacker knows IV, padding, and controls plaintext (called padding oracle attack!)



Cipher Block Chaining (CBC) mode encryption

Cipher Properties

- Encryption and Decryption are inverse properties
  - $m = D_k(E_k(m))$
- Informally: ciphertext should reveal **nothing** about the plaintext
- Non-property: integrity
  - May be possible to change decrypted plaintext in known way
  - Needs separate message authentication
- Key hygiene
  - **Do not use the same key with different modes** (or for separate encryption or authentication operations!)

So, that's symmetric cryptography…

- We can now protect confidentiality and integrity of messages without sharing very large secrets.
- But it assumes we can easily share keys with each other.
    - **Turns out this is very hard to do in practice!**
- And that's why we have….

Asymmetric Cryptography

Asymmetric Cryptography

- aka ***Public Key Cryptography***
- Two separate keys: ***public key*** and ***private key*** (secret)
- Public key ***known to everyone***
  - Given Alice's public key
    - Anyone can send an encrypted message to Alice
    - Anyone can verify that a message was signed by Alice
- Private key is kept secret
  - Only Alice can decrypt messages encrypted with her public key
  - Only Alice can sign messages so that they can be verified with her public key

Asymmetric Cryptography

- Examples?

Asymmetric Cryptography

- Examples?
  - SSH keys! Very common usage of asymmetric cryptography that you use all the time!
  - HTTPS keys
  - TLS… we'll talk about this in two lectures :)

Asymmetric Primitives

- Confidentiality: encryption and decryption
- Integrity and Authenticity: signing and verification

Asymmetric Cryptography Notation

- Keys are generated with a *key generation function*
  - Keys must have mathematically inverse properties (again, we won't get into this… but it's cool math!)
- Notation:
  - *K*: public key
  - *k:* private key
  - *r:* random bits

(K, k) <— Keygen(r)

Asymmetric Encryption and Decryption

- Encryption uses public key

$$c = E_K(m)$$

- Decryption uses private key

$$m = D_k(c)$$

- Computationally hard to decrypt without private key. Messages are fixed size. In general, **computationally slow** (relative to symmetric cryptography)

Asymmetric Usage

- Public directory contains everyone's public key
- To encrypt to a person, get their public key from the directory
- **No need for shared secrets!** (in theory….)
- In practice, we use asymmetric key cryptography to do *key exchange* (we'll talk about this next time) and then use *symmetric key cryptography* for session-level communication

Asymmetric Signing and Verification

- Signing uses private key

$$s = S_k(m)$$

- Verification uses public key

$$v = V_K(m, s)$$

- Computationally hard to sign without private key. Message again are fixed size.

Testing your understanding

- Alice wants to send an encrypted message **m** to Bob. What should she send to him?

Testing your understanding

- Alice wants to send an encrypted message **m** to Bob. What should she send to him?
    - Send **c = Bob_E$_K$(m)**
- Bob wants to verify that a message **m** and signature **s** came from Alice. What should he do?

Testing your understanding

- Alice wants to send an encrypted message **m** to Bob. What should she send to him?
  - Send **c = Bob_E$_K$(m)**
- Bob wants to verify that a message **m** and signature **s** came from Alice. What should he do?
  - Check that **s = Alice_V$_K$(m)**
- Alice wants to sign a message **m** to Bob so Bob knows it came from her. What should she send him?

Testing your understanding

- Alice wants to send an encrypted message **m** to Bob. What should she send to him?
    - Send **c = Bob_E_K(m)**
- Bob wants to verify that a message **m** and signature **s** came from Alice. What should he do?
    - Check that **s = Alice_V_K(m)**
- Alice wants to sign a message **m** to Bob so Bob knows it came from her. What should she send him?
    - Send **m, Alice_S_K(m)**

Combining Authentication and Encryption

- Three major schemes:
  - Encrypt-then-MAC: **MAC(Enc(m))**
  - MAC-then-Encrypt: **Enc(MAC(m))**
  - Encrypt-and-MAC: **Enc(m), MAC(m)**
- Encrypt-then-MAC offers the *most* security (use this), because it preserves *integrity of the ciphertext* and *plaintext integrity*
  - MAC-then-Encrypt doesn't offer integrity on the ciphertext; we have no way of knowing until decryption if the message was authentic
  - Encrypt and MAC doesn't offer ciphertext integrity because MAC is taken on plaintext

Classic Asymmetric Ciphers

- ElGamal encryption (1985)
  - Based on Diffie-Hellman key exchange (1976)
  - Computational basis: **hardness of discrete logarithm**
  - For large prime p, generator g, and value h, it's easy to compute h = g^n (modp) but hard to find exponent n given h.
- RSA encryption (1978)
  - Rivest, Shamir, Adleman
  - Computational basis: **hardness of factoring** (integer factoring product of large primes is computationally intractable)

Classic Asymmetric Signatures

- DSA: Digital Signature Algorithm (1991)
  - Closely related to ElGamal signature scheme
  - Computational basis: **hardness of discrete logarithm**
- RSA signatures
  - Computational basis: **hardness of factoring** (integer factoring product of large primes is computationally intractable)

Practical considerations

- Asymmetric cryptography operations generally **much** more expensive than symmetric operations
  - Compute, key size, you name it
- Asymmetric primitives operate on fixed-size messages
- **Never use the same keypair for the same activity.** Use different keypairs for encryption / signatures!
- Usually combined with symmetric crypto for performance
  - e.g., use asymmetric keys to bootstrap and ephemeral one-time key for symmetric encryption

Combined symmetric / asymmetric encryption

- **Encryption:**
  - Generate an ephemeral (one time) symmetric key (i.e., random)
  - Encrypt message using this ephemeral key
  - Encrypt ephemeral key using asymmetric encryption
  - Send encrypted message and encrypted ephemeral key
- **Decryption:**
  - Decrypt ephemeral key, use it to decrypt message

Combined symmetric / asymmetric signing

- **Signing:**
  - Compute cryptographic hash of message and sign it using asymmetric signature scheme
- **Verification:**
  - Compute cryptographic hash of message and verify it using asymmetric signature scheme

Review

- Confidentiality and integrity are protected by different cryptographic mechanisms
    - Having one does not imply the other!!!!
- Kerckhoff's Principle: Do not rely on security by obscurity; don't use secret functions, use secret *keys*
- Big one: **DO NOT ROLL YOUR OWN CRYPTO.** Use existing methods and tools. It is very easy to shoot yourself in the foot…
    - Do not descend into lower layers unless you're an expert
    - Do not modify or re-implement cryptographic libraries…. minor changes can lead to catastrophic failures!

Next time…

- We talk about key exchange, forward secrecy, and get into the weeds of RSA
- Good luck on PA4!