

Crypto Part 1

(and Software Defenses Wrap-up)

CMSC 23200, Spring 2025, Lecture 4

Grant Ho

University of Chicago

Logistics

Assignment 1: **Both parts** Due **TONIGHT** by 11:59pm

- Part A: Gradescope
- Part B: Gradescope + Canvas

Assignment 2 (Buffer Overflow Attacks): Released on Friday afternoon

- For **Assignment 2 only**, you will use a ***different course VM*** (with a different hostname): read the assignment instructions for details

Outline: Crypto + Software Security Wrap-up

1. Memory Safety Defenses

- Fuzzing
- Memory Safe Languages

2. Crypto Part 1: Symmetric Key Cryptography

Program Fuzzing: Find bugs before release

Idea: Developer runs their program on huge number of automatically-generated inputs, searches for crashes, and fixes bugs before releasing software

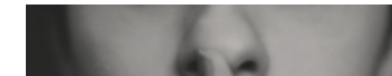
Linux Mint fixes screensaver bypass discovered by two kids

Two children playing on their dad's computer accidentally found a way to bypass the screensaver and access locked systems.



By [Catalin Cimpanu](#) for [Zero Day](#) | January 15, 2021 -- 18:28 GMT (10:28 PST) | Topic: [Security](#)

MORE FROM CATALIN CIMPANU



Security
Hacker leaks data of

"A few weeks ago, my kids wanted to hack my Linux desktop, so they typed and clicked everywhere while I was standing behind them looking at them play," wrote a user identifying themselves as robo2bobo.

According to the bug report, the two kids pressed random keys on both the physical and on-screen keyboards, which eventually led to a crash of the Linux Mint screensaver, allowing the two access to the desktop.

"I thought it was a unique incident, but they managed to do it a second time," the user added.

Two Types of Fuzzing Strategies

Mutation-based (dumb): Take an initial set of examples (program inputs) and make random changes to them.

- Millions of inputs (can run fuzzing forever)
- Possibly lower quality, unlikely to find certain bugs / types of inputs

Generative (smart): Describe inputs to fit format/protocol, then generate inputs from that grammar with changes.

- Run with fewer inputs, which can be directed to certain bug types or code logic

Problems with Fuzzing

Mutation-based (dumb): How long to run? And we need a strong server.

Generative (smart): Run out of test cases. A lot more work.

General problems:

- Need to identify when bug/crash occurs automatically.
- Don't want to report same bug 1000s of times.
- How do we prioritize bugs?

Fuzzing in Production

AFL: Popular open-source fuzzer released by Google

Google/Microsoft constantly fuzz products with dedicated servers/VMS.

Anecdote: Found 95 vulnerabilities in Chrome during 2011.



OneFuzz

A self-hosted Fuzzing-As-A-Service platform

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a [single command](#), which can be [baked into CI/CD](#), developers can launch fuzz jobs from a few virtual machines to thousands of cores.

Memory-Safe Languages

Many of our problems can be solved by using “memory-safe” languages.

- The programming model for these languages *does not allow* for such bugs (e.g., no access to pointers / mem addr’s and built-in object bounds checking).

Not Memory-Safe	Memory Safe
C	Java
C++	Python
Assembly	Javascript
	Rust, Go, Haskell, ...

Ideally, we’d avoid writing programs in unsafe languages, but lots of legacy code (and low-level stuff) are written in C/C++.

Recap: Software Defenses

Pre-deployment, before the program runs: find or prevent bugs

- Fuzzing: proactively finding & fixing bugs by testing many program inputs
- Memory safe languages: automatically avoid exploitable memory bugs
- Done by [the application developer](#)

Program runtime: stopping exploits / violations of program's memory

- Stack Canaries, ASLR, DEP/W+X, etc.
- Implemented by the [compiler \(stack canary\)](#) or [operating system \(ASLR, W+X\)](#)
- Attacks adapt & evolve (Stack reading, ROP attacks, etc.)

Post-exploitation (not covered): limit possible damage from compromise

- Sandboxing and VMs
- Done by [user/admin of the system](#) or [the app developer](#) (e.g., web browsers)

Cryptography: Part 1

(Slides adapted from David Cash and Dan Boneh)

Outline: Cryptography Part 1

1. Memory Safety Defenses

- Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography

- Common goals & Threat models
- Encryption & Basic ciphers
- One-time pads (Theoretical Encryption)
- Stream ciphers (Practical Encryption Tool #1)
- Block ciphers (Practical Encryption Tool #2)

What is Cryptography (for CMSC 23200)?

Cryptography develops algorithms that achieve security goals (CIA).

Cryptography involves using math / theory to stop adversaries.

This Course:

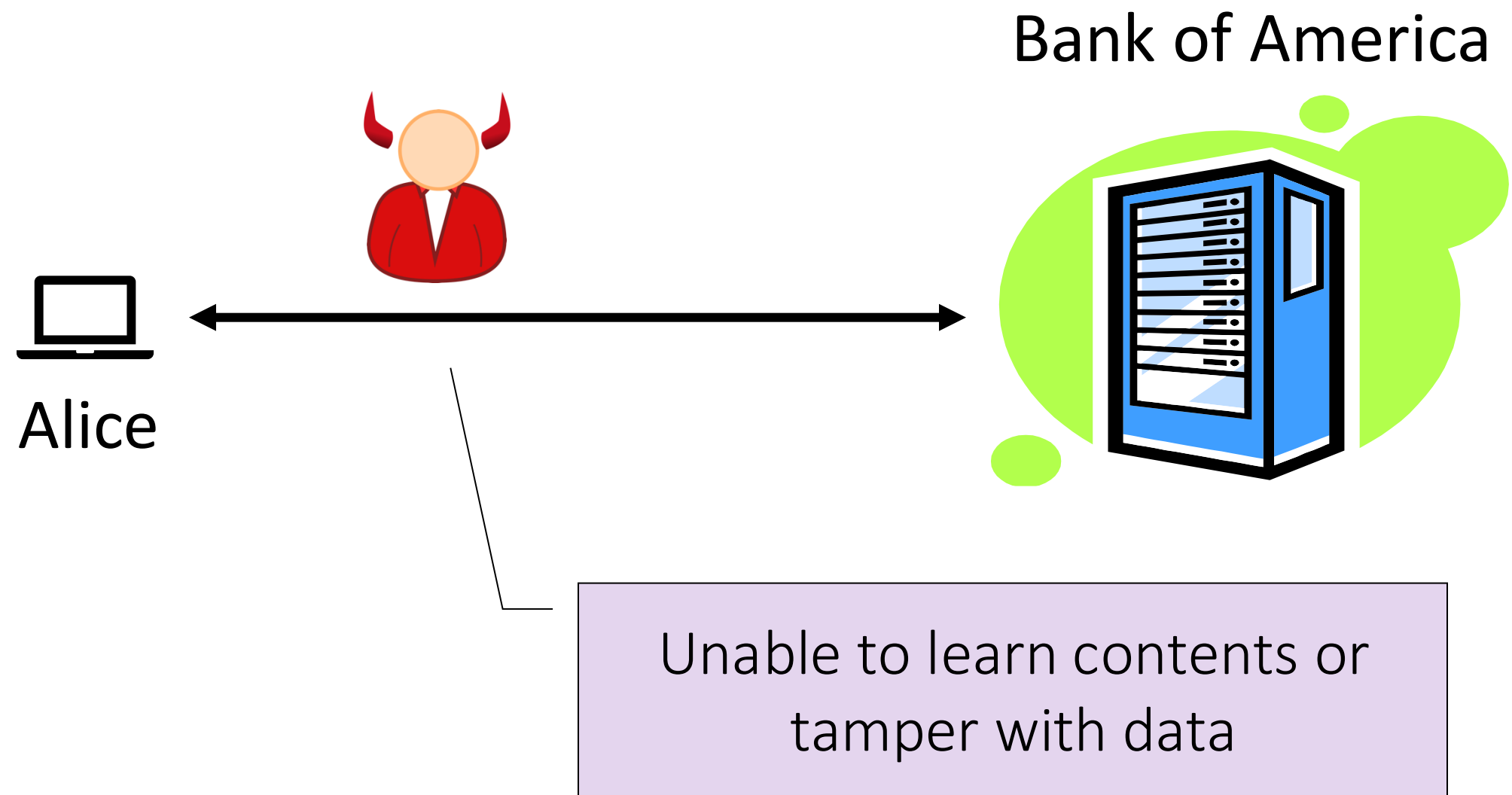
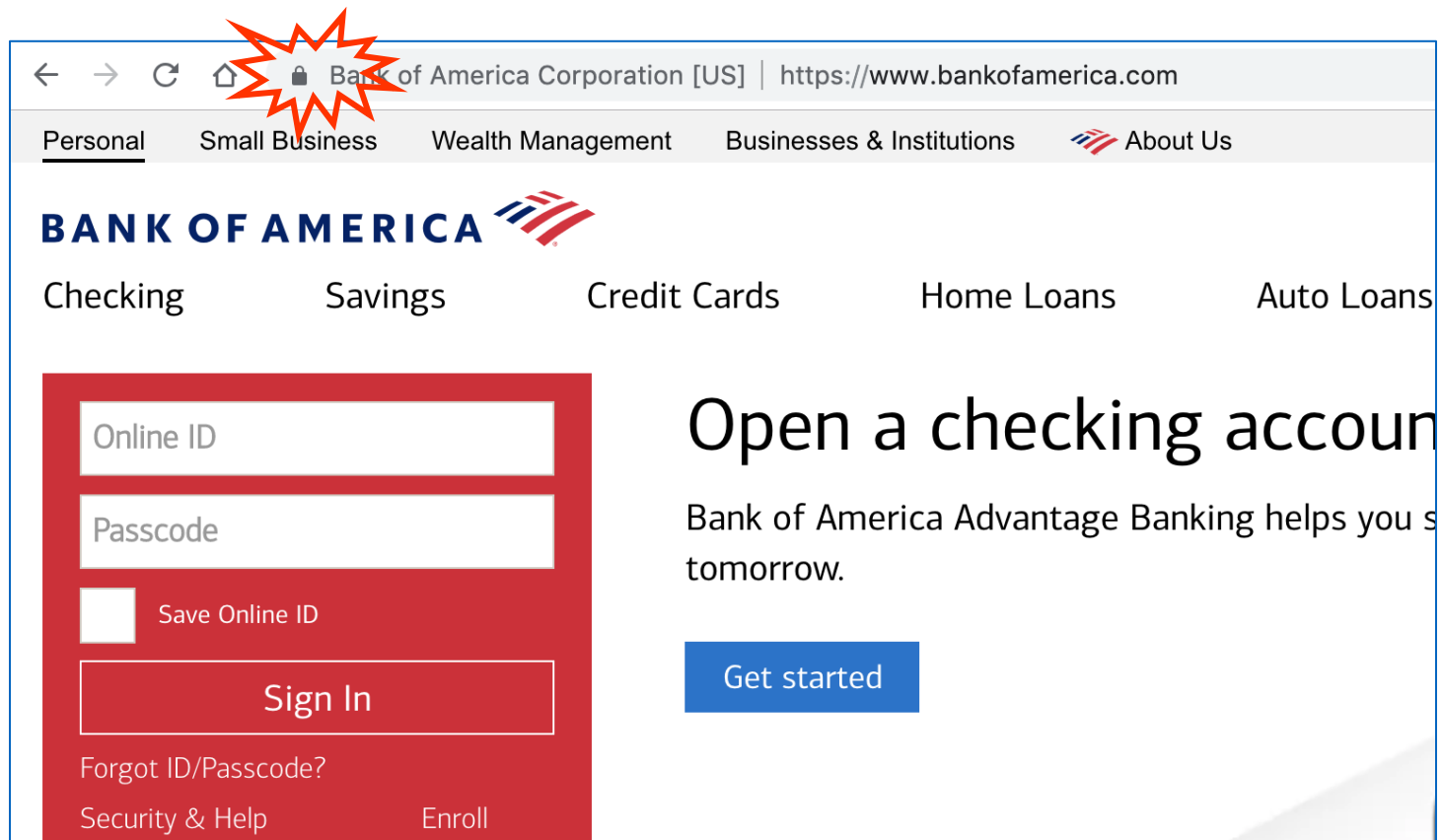
- A brief overview of major crypto concepts and tools
- Cover (some) big “gotchas” in crypto deployments
- Not going to cover math, proofs, or many theoretical details.
Consider taking CS284 (Cryptography)!

Common High-Level Goal: Create a Secure Channel

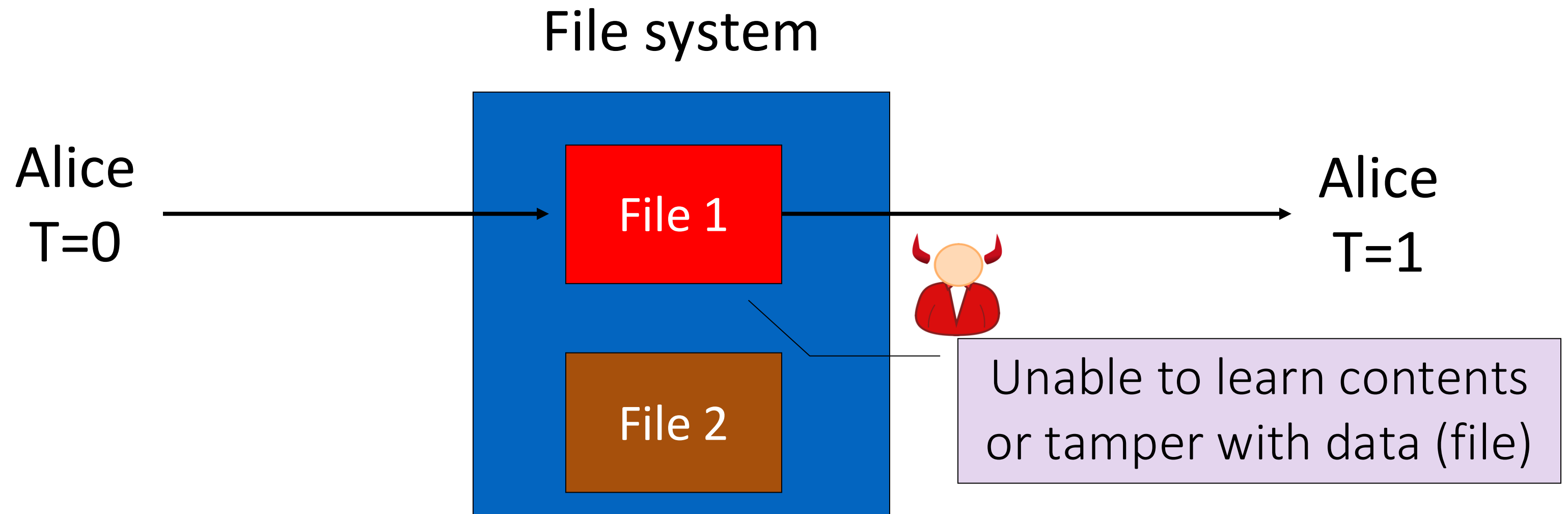


Goal: Attacker does not learn anything about the contents of messages and cannot tamper with their contents.

Example 1: Secure communication (protecting data in motion)



Example 2: Protected files (protecting data at rest)



Three Key Security Goals of Cryptography

- 1. Confidentiality:** an attacker cannot learn the contents of our data
- 2. Integrity:** an attacker cannot modify the contents of our data
- 3. Authentication:** an attacker cannot masquerade as someone else, or make us believe their message/data was sent by someone else

Four Cryptography Problems / Tools

Security Goal	Confidentiality	Authenticity/Integrity
Pre-shared key?	Yes ("Symmetric")	
No ("Asymmetric")		

Four Cryptography Problems / Tools

Security Goal	Confidentiality	Authenticity/Integrity
Pre-shared key?	Symmetric Encryption	Message Authentication Code (MAC)
No ("Asymmetric")		

Four Cryptography Problems / Tools

Security Goal	Confidentiality	Authenticity/Integrity
Pre-shared key?	Symmetric Encryption	Message Authentication Code (MAC)
No ("Asymmetric")	Public-Key Encryption	Digital Signatures

Outline: Cryptography Part 1

1. Memory Safety Defenses

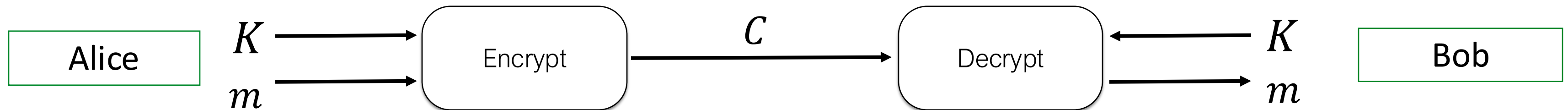
- Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography

- Common goals & Threat models
- Encryption & Basic ciphers
- One-time pads (Theoretical Encryption)
- Stream ciphers (Practical Encryption Tool #1)
- Block ciphers (Practical Encryption Tool #2)

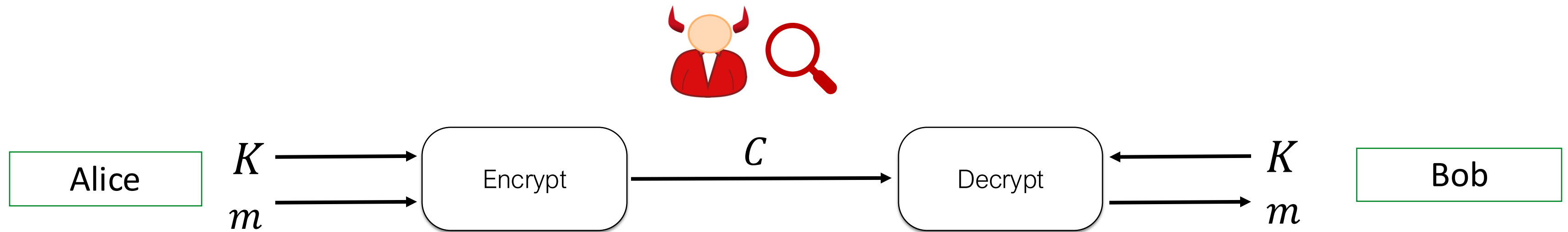
Ciphers (a.k.a. Symmetric Encryption)

A cipher is a pair of algorithms Encrypt, Decrypt:



- **Encryption** algorithm: $\text{Encrypt}(K, m) = c$
 - Convert a plaintext message m , into an encrypted message c (ciphertext)
- **Decryption** algorithm: $\text{Decrypt}(K, c) = m$
 - Convert a ciphertext c , back into its plaintext message m

Encryption: Providing Confidentiality



Threat Model: **Passive attacker**

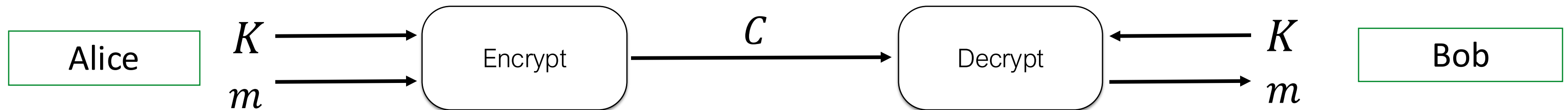
- Adversary see the ciphertexts, but they **cannot** modify them in any way
- Attacker's goal: learn something about plaintext messages from ciphertexts

Today's Lecture: **Symmetric key setting:**

- Alice & Bob already have a shared secret key, K , that the attacker does not know

Ciphers (a.k.a. Symmetric Encryption)

A cipher is a pair of algorithms Encrypt, Decrypt:



Requirements of a Secure Cipher:

- **Correctness:** decryption recovers the same message.
 - $\text{Encrypt}(K, m) = c$ and $\text{Decrypt}(K, c) = m$
- **Confidentiality (Security):** the ciphertext c reveals nothing about the message m (other than the message length)

Evaluating Security of Crypto Algorithms

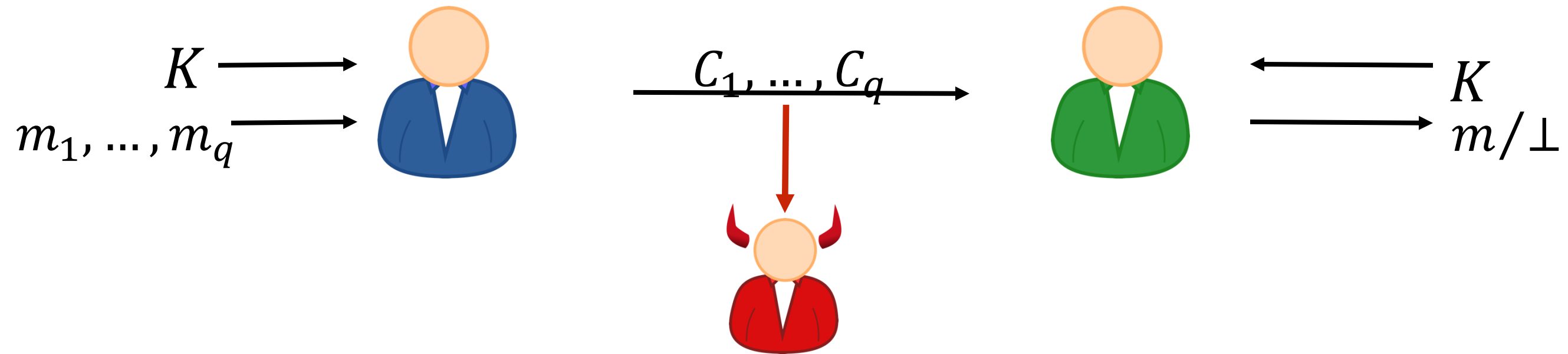
Kerckhoff's Principle:

Assume the adversary knows your algorithms and implementation.
The only thing they don't know is the key.

Example:

- Adversary knows Alice & Bob using SSH, and they know logic/code of all the ciphers that SSH allows (e.g., by downloading the open-source software itself)
- But they do *not* know the keys that Alice & Bob use

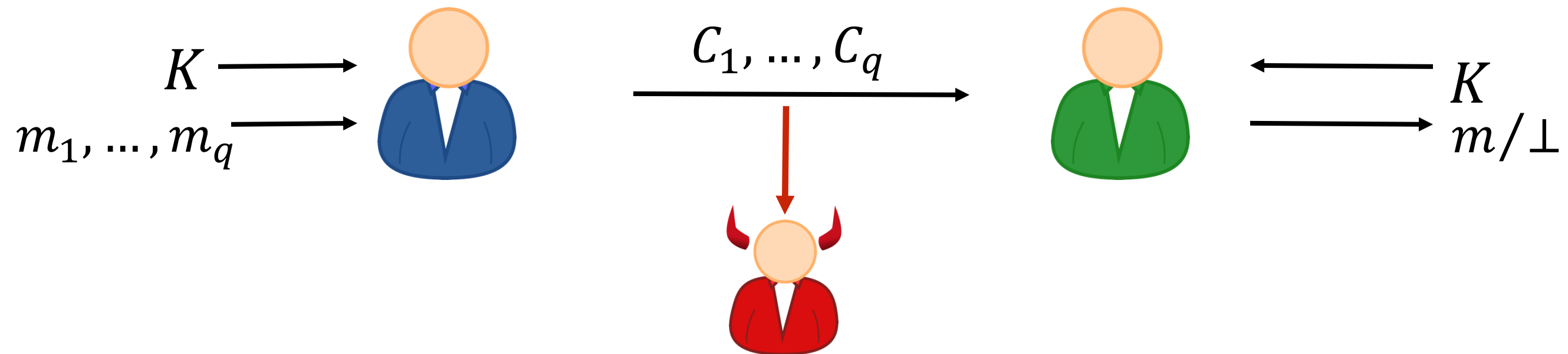
Adversary Goal: Break Confidentiality



The adversary sees ciphertexts and attempts to recover some “useful information” about plaintexts.

Other attack settings are important
(e.g. adversary can ask for some encryptions, some decryptions...)

Attacks can succeed without recovering the key

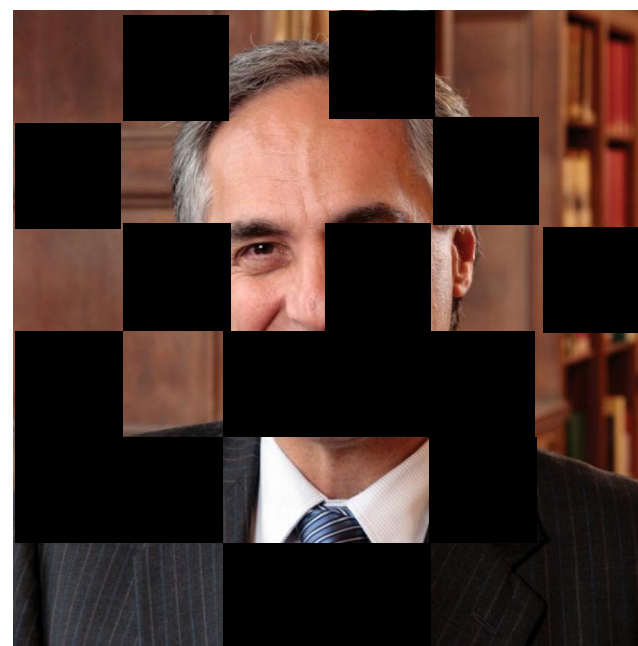


Full break: Adversary recovers K , decrypts all ciphertexts.

However: Clever attackers may learn plaintext information from ciphertexts without recovering the key.
If so, the attack has succeeded / encryption algorithm is insecure.

Partial Knowledge & Recovering Partial Information

- Recovering entire messages is useful
- But recovering **partial information** is also be useful & dangerous



A lot of information is missing here.

But can we say who this is?

- Attacker may know large parts of plaintext already (e.g. formatting strings or application content).

The attacker tries to obtain something it doesn't already know.

M = `http://site.com?password=` ■ ■ ■ ■ ■ ■ ■ ■

Secure Encryption Goal

An **attack** is successful as long as it recovers any new info about the plaintext (caveat: typically ignore length of text).

Secure Encryption must hide all information about plaintexts (including any possible / partial information).

- Ciphertext reveals nothing about its plaintext message

Historical Cipher: ROT13 (“Caesar cipher”)

Encrypt(K,m): shift each letter of plaintext forward by K positions in the alphabet (wrap from Z to A).

Plaintext: DEFGH

Key (shift): 2

Ciphertext: FGHKL

Plaintext: ATTACKATDAWN

Key (shift): 13

Ciphertext: NGGNPXNGQNJJA

Historical Cipher: Substitution Cipher

Encrypt(K,m): The key K is a permutation π on $\{A, \dots Z\}$.

Apply π to each character of m to create c

M: ATTACKATDAWN

K: π 

C: ZKKZAMZKYZGT

x	$\pi(x)$
A	Z
B	U
C	A
D	Y
E	R
F	E
G	X
H	B
I	D
J	C
K	M
L	Q
M	H
N	T
O	I

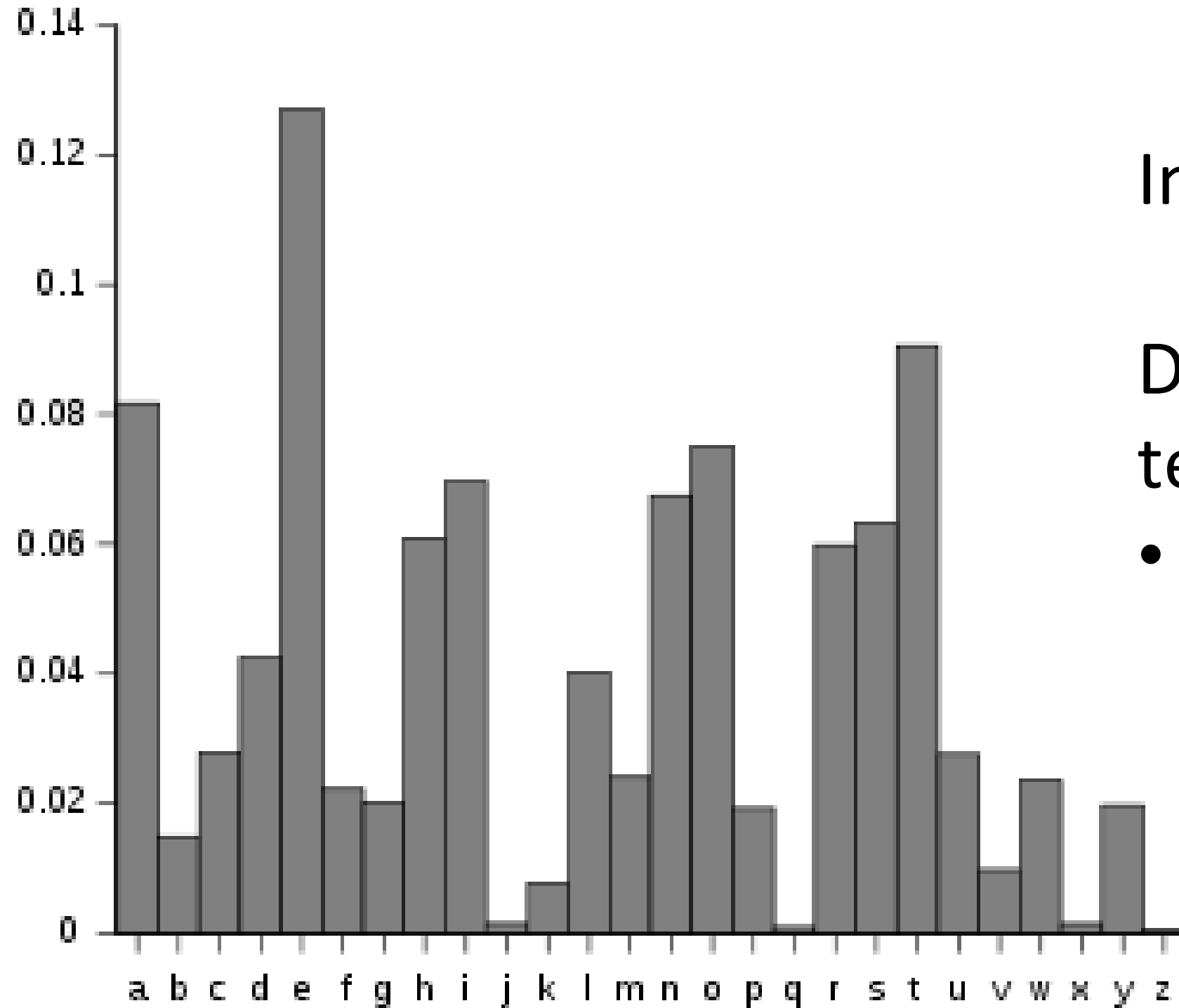
How many keys?

$$26! \approx 2^{88}$$

9 million years to try all keys at rate of 1 trillion/sec

Q: Is this secure?

Cryptanalysis of Substitution Cipher



Insecure!

Distribution of letters in English text is not uniform:

- Can guess letters in a long msg by computing their frequency

Outline: Cryptography Part 1

1. Memory Safety Defenses

- Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography

- Common goals & Threat models
- Encryption & Basic ciphers
- One-time pads (Theoretical Encryption)
- Stream ciphers (Practical Encryption Tool #1)
- Block ciphers (Practical Encryption Tool #2)

Quick recall: Bitwise-XOR operation

We will use bit-wise XOR:

$$\begin{array}{r} 0101 \\ \oplus 1100 \\ \hline 1001 \end{array}$$

Some Properties:

- $X \oplus Y = Y \oplus X$
- $X \oplus X = 000\dots 0$
- $X \oplus Y \oplus X = Y$

Cipher: One-Time Pad (OTP)

Key K: Bitstring of length L

Plaintext M: Bitstring of length L

Encrypt(K,M): Output $K \oplus M$

Decrypt(K,C): Output $K \oplus C$

Example:

$$\begin{array}{r} 0101 \quad (K) \\ \oplus 1100 \quad (M) \\ \hline 1001 \quad (C) \end{array}$$

Correctly decrypts because

$$K \oplus C = K \oplus (K \oplus M) = (K \oplus K) \oplus M = M$$

Q: Is the one-time pad secure? Yes*

Security of the One-Time Pad (OTP)

If key is random & used only once, then OTP provides confidentiality.

- “Proof”: if an adversary sees **only one** ciphertext using a random key, then any plaintext is equally likely, so they cannot recover any partial information besides the plaintext length.

Ciphertext observed: 10111
Possible plaintext: 00101
⇒ Possible key: 10010

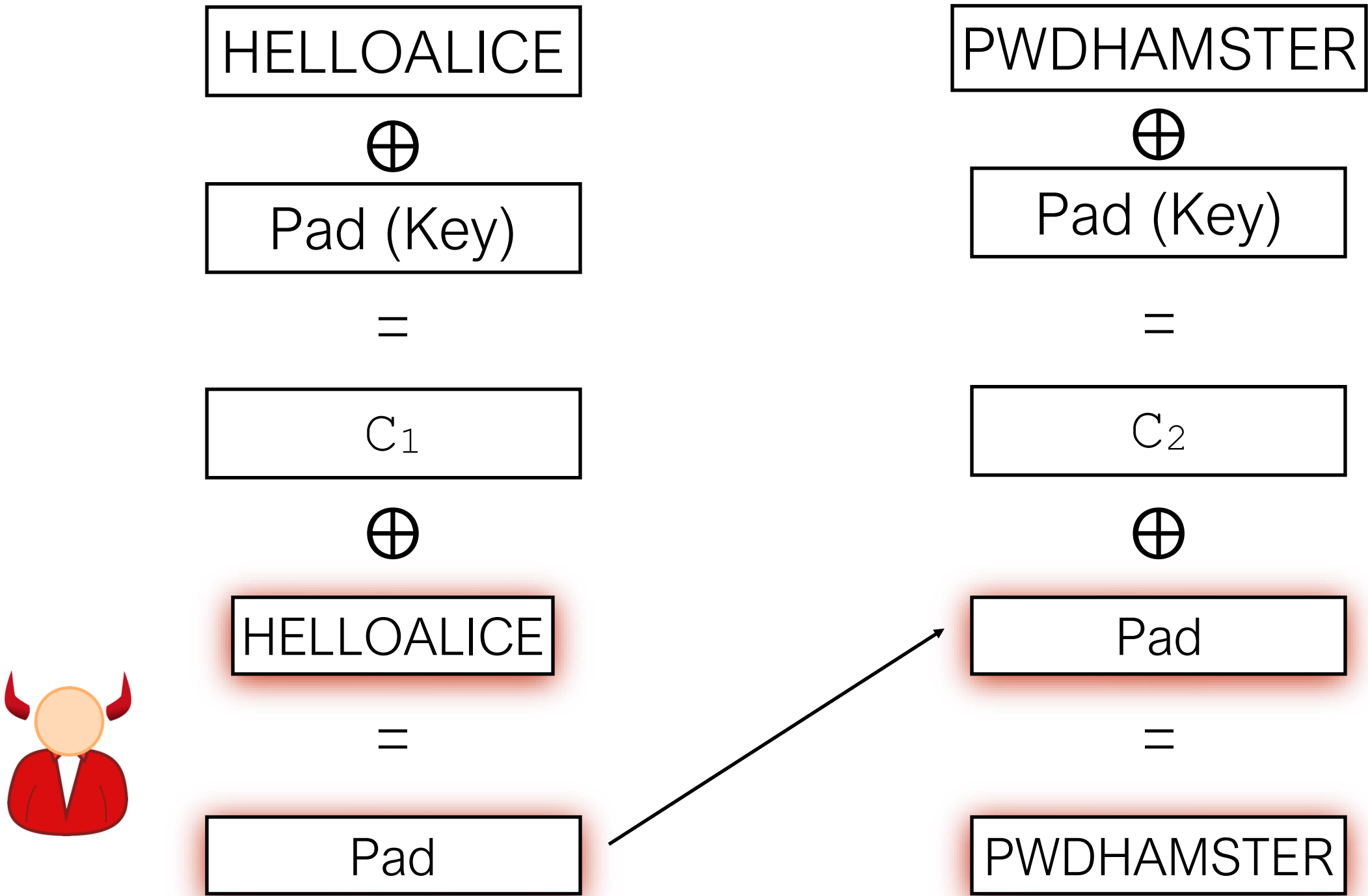
Ciphertext observed: 10111
Possible plaintext: 11111
⇒ Possible key: 01000
(equal likelihood)

1. Adversary goal: Learn partial information from plaintext
2. Adversary capability: Observe a single ciphertext
3. Adversary compute resources: Unlimited time/memory (!)

Issues with One-Time Pad (OTP)

1. Reusing a pad is insecure
2. One-Time Pad has a long key

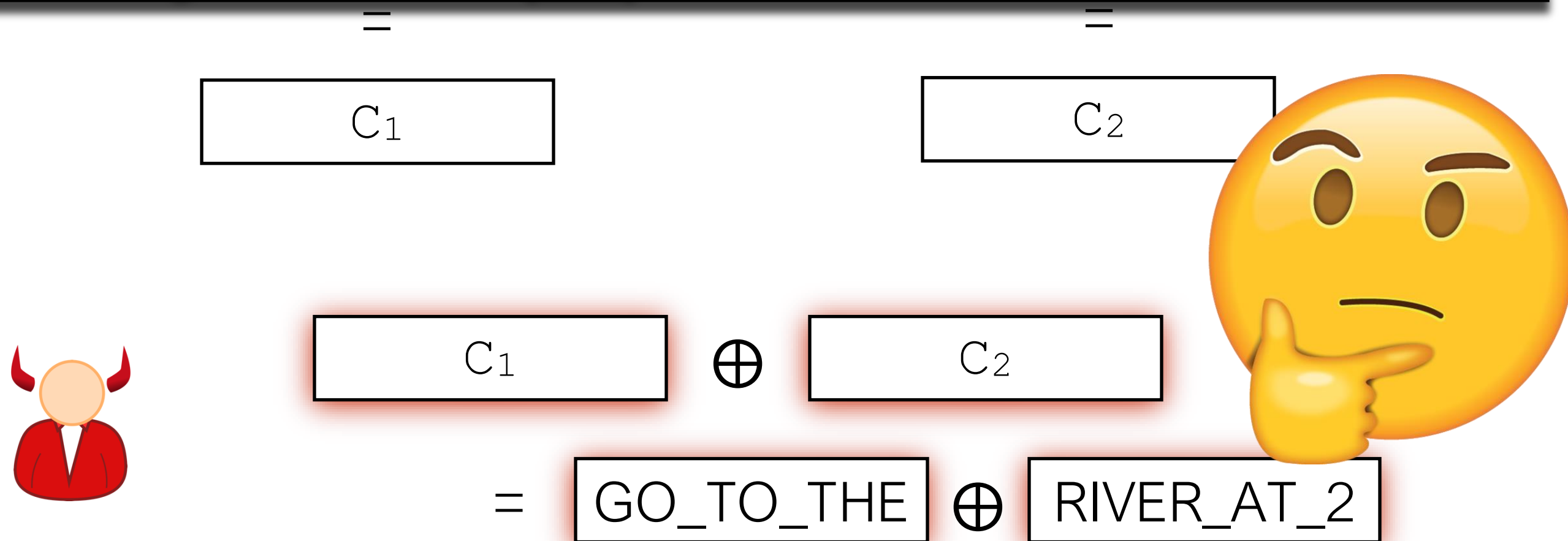
Issue #1: Reusing a One-Time Pad is Insecure



Issue #1: Reusing a One-Time Pad is Insecure

Has led to real attacks:

- Project Venona (1940s) attack by US on Soviet encryption
- MS Windows NT protocol PPTP
- WEP (old WiFi encryption protocol)
- Fortiguard routers! [[link](#)]



Issue #2: One-Time Pad Needs a Long Key

By definition: OTP needs $\text{Key-length} \geq \text{Plaintext-length}$

- Long message = long key required
- If we could've securely shared the key (one-time pad), then we could've just securely shared the plaintext!
- Not realistic to use in practice

Outline: Cryptography Part 1

1. Memory Safety Defenses

- Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography

- Common goals & Threat models
- Encryption & Basic ciphers
- One-time pads (Theoretical Encryption)
- Stream ciphers (Practical Encryption Tool #1)
- Block ciphers (Practical Encryption Tool #2)

Stream Ciphers: Simulating OTPs

Key Idea: Given a random key, K , create an extremely large pseudo-random string that can be used as a one-time pad

- Cryptographic functions called pseudo-random number generators (PRNGs) that can do this

Stream Cipher Security Goal (Sketch)

Security goal: When k is random and unknown, $G(k)$ should “look” random.

... even to an adversary spending a lot of computation.

Much stronger requirement that “passes statistical tests”.

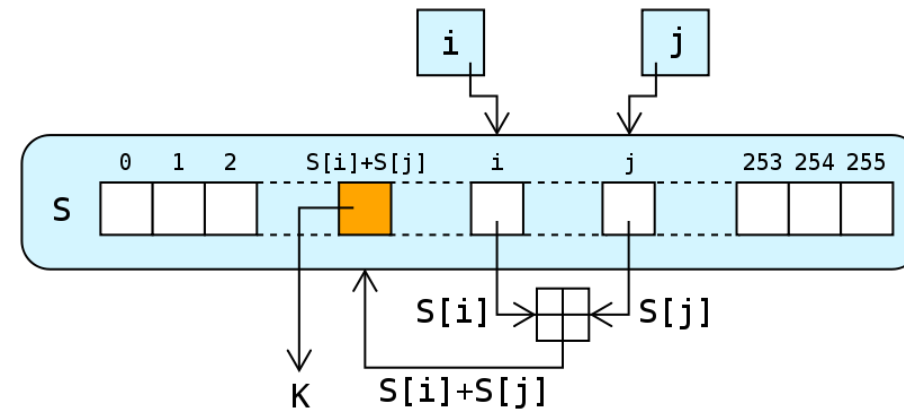
Brute force attack: Given $y=G(k)$, try all possible k and see if you get the string y .

Clarified goal: When k is random and unknown, $G(k)$ should “look” random to anyone who can't run a brute force attack.

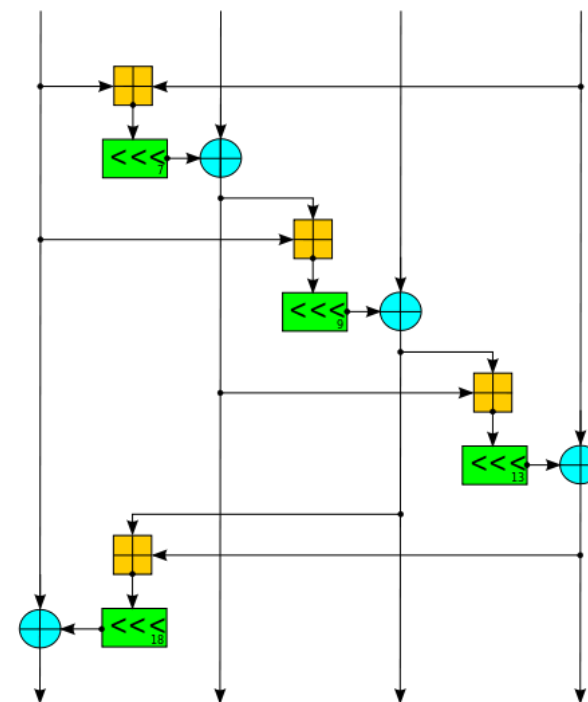
(key length = 256-bits is considered strong now)

Practical Stream Ciphers (Not covered in this class)

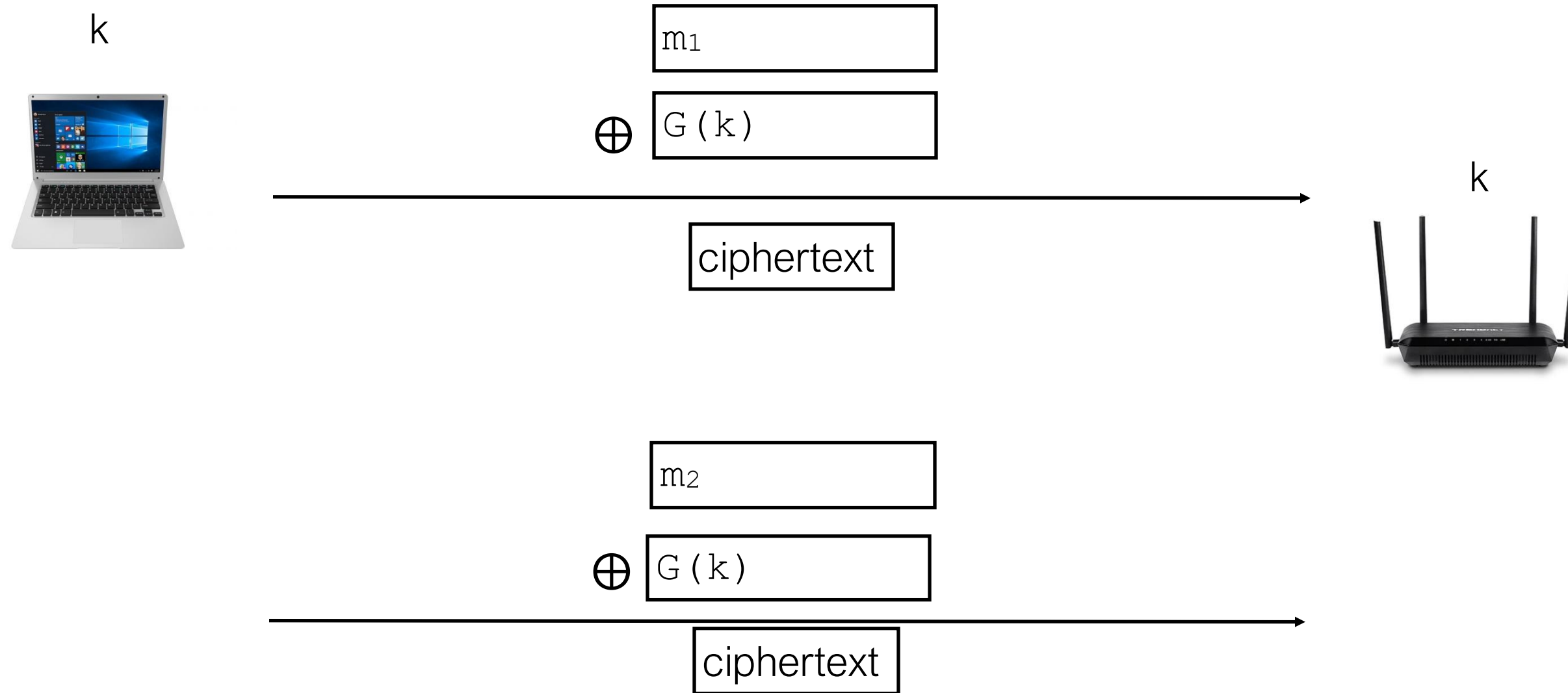
RC4 (1987): “Ron’s Cipher #4”. Mostly retired by 2016 (insecure).



ChaCha20 (2007): Successfully deployed replacement.
Supports *nonces*.



Sending Multiple Messages w/ Stream Ciphers: Pad Reuse?



...

Uh oh... two-time pad!

Addressing pad reuse: Stream cipher with a nonce

Stream cipher with a nonce: Algorithm G that takes **two inputs** and produces a very long bit-string as output.

<u>Nonce IV:</u>	<u>Key/Seed k:</u>
1100...11	1100...11

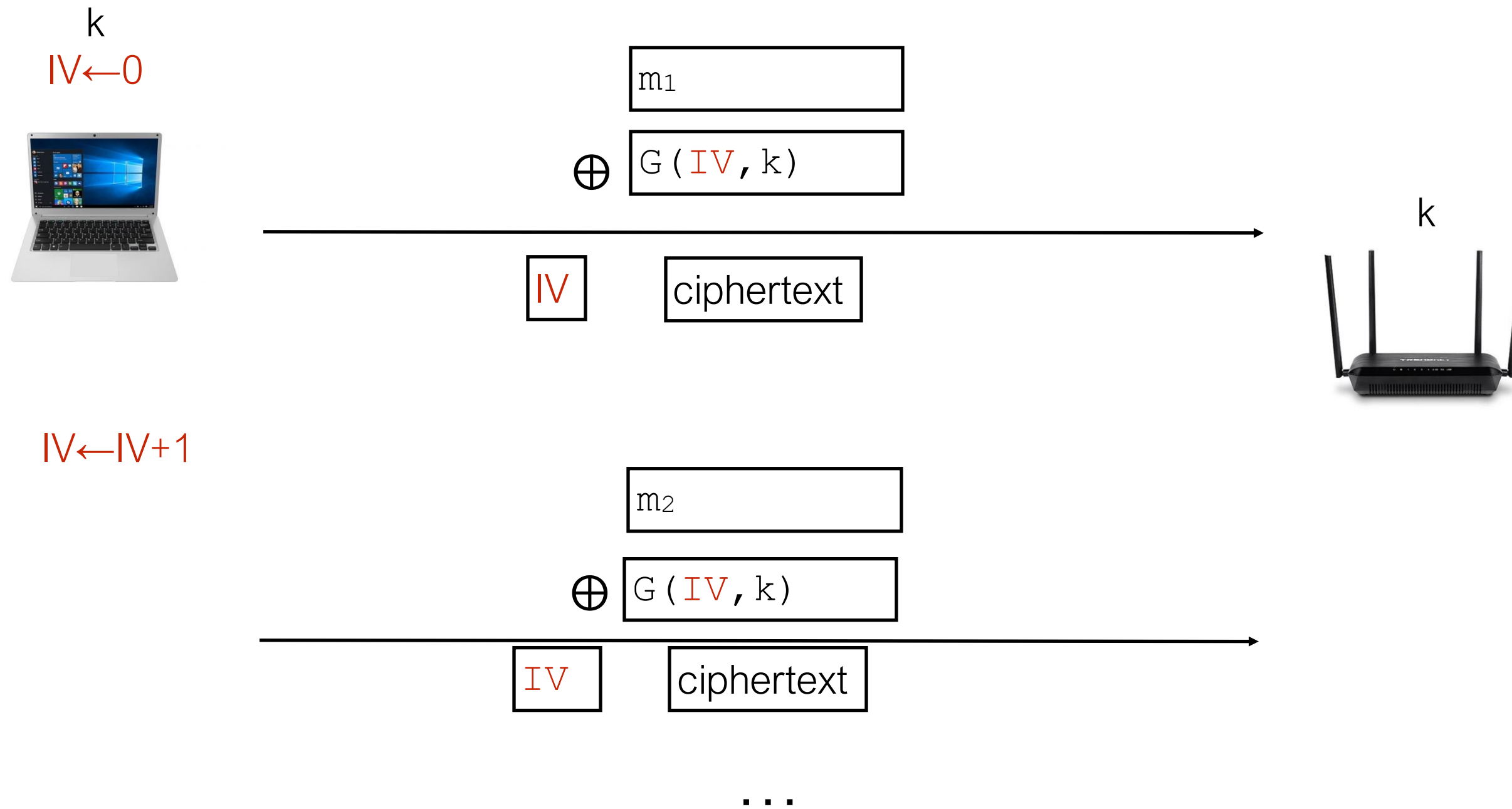


$G(IV,k)$: 11111010001000111010100101000101100100111100...

- “nonce” = “number once”.
- Usually denoted IV = “initialization vector”

Security: When k is random and unknown, $G(IV, k)$ should “look” random and independent for each value of IV .

Solution: Stream cipher with a nonce



- However: if nonce repeats, then pad repeats -> two-time pad!

Example of Pad Re-use: WEP



Warning: Broken



IEEE 802.11b WEP: WiFi security standard '97-'03

IV



IV is 24-bit wide counter

- Repeats after 2^{24} frames (≈ 16 million)
- IV is often set to zero on power cycle

Solutions: (WPA2 replacement)

- Larger IV space, or force rekeying more often
- Set IV to combination of packet number, address, etc

Example of Pad Re-use: WEP



Warning: Broken



IEEE 802.11b WEP: WiFi security standard '97-'03

- Re
- Of



The screenshot shows the top of an Ars Technica article. The header includes the 'ars TECHNICA' logo and navigation links for 'BIZ & IT', 'TECH', 'SCIENCE', 'POLICY', 'CARS', 'GAMING & CULTURE', and 'FORUMS'. The article title is 'Serious flaw in WPA2 protocol lets attackers intercept passwords and much more'. Below the title, it states 'KRACK attack is especially bad news for Android and Linux users.' and is dated 'DAN GOODIN - 10/15/2017, 11:37 PM'.

- Solutions: (W)
- Larger IV sp
 - Set IV to combination of packet number, address, etc

parameters to their initial values. KRACK forces the nonce reuse in a way that allows the encryption to be bypassed. Ars Technica IT editor Sean Gallagher has [much more about KRACK here](#).

Stream Ciphers w/ Nonces

Stream ciphers use PRNG's & nonces to make secure one-time pads practical.

1. Reusing a pad is insecure  *Use unique nonces*
2. One-Time Pad needs a long key  *Use stream cipher with short key*

Outline: Cryptography Part 1

1. Memory Safety Defenses

- Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography

- Common goals & Threat models
- Encryption & Basic ciphers
- One-time pads (Theoretical Encryption)
- Stream ciphers (Practical Encryption Tool #1)
- Block ciphers (Practical Encryption Tool #2)

Block Ciphers (AES) : Another Tool for Secure Encryption

Blockciphers: common crypto building block for solving many problems.

Informal definition: A blockcipher is essentially a substitution cipher with a very large alphabet and a very compact key.

Typical parameters:

Alphabet = $\{0, 1\}^{128}$ (16 bytes input -> 16 bytes output)

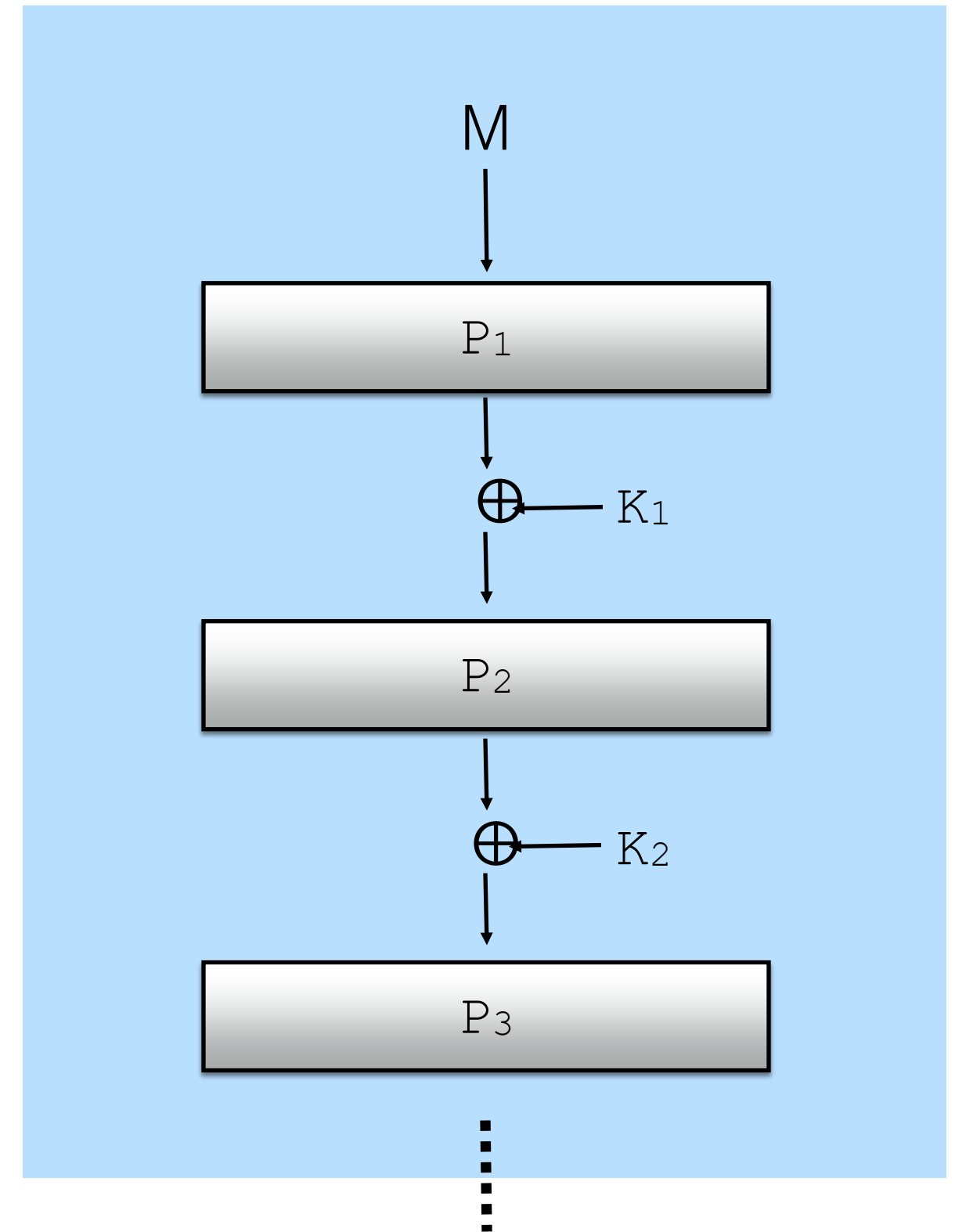
Key length = 16 bytes (2^{128} possible key values)

Can build many higher-level protocols from a good blockcipher.

Advanced Encryption Standard (AES)

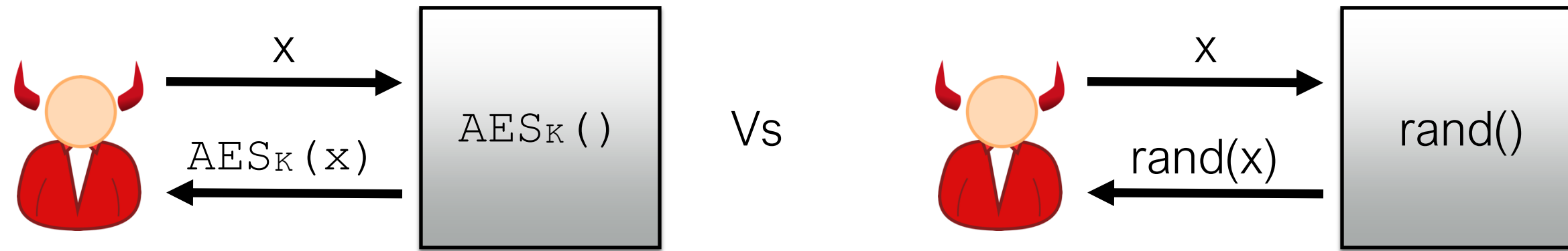
- NIST ran competition to develop standard encryption algorithms in 1997
- Several submissions, *Rijndael (AES)* chosen and standardized as the secure block cipher design

- Rijmen and Daemen
 - Block length $n = 128$
 - Key length $k = 128, 192, 256$
 - 10 rounds of “substitution-permutation network”
- Break msg M into blocks and encrypt each block



Blockcipher Security (Confidentiality)

- AES is thought to be a good “Pseudorandom Permutation” (PRP)



- Outputs all look random and independent for different inputs, even when inputs are maliciously controlled.
- Formal definition in CS284.

Example - AES Input/Outputs

Keys and inputs are 16 bytes = 128 bits

-K1 : 9500924ad9d1b7a28391887d95fcfbd5

-K2 : 9500924ad9d1b7a28391887d95fcfbd6

$AES_{K1}(00 \dots 00) = 8b805ddb39f3eee72b43bf95c9ce410f$

$AES_{K1}(00 \dots 01) = 9918e60f2a20b1b81674646dceebdb51$

$AES_{K2}(00 \dots 00) = 1303270be48ce8b8dd8316fdbba38eb04$

$AES_{K2}(00 \dots 01) = 96ba598a55873ec1286af646073e36f6$

AES is now the gold standard block-cipher

- Very fast; Intel & AMD CPU chips have built-in AES instructions

Block Cipher Modes

AES only encrypts 16 bytes at a time

- Question: What do I do if I want to encrypt more than 16 bytes of data?
- Answer: AES has different *modes* of operation
 - Some common modes: ECB, CTR, CBC, GCM
 - ECB : do not use – insecure!!
 - CTR & CBC : confidentiality, but not integrity
 - GCM: authenticated encryption (next week)

ECB Mode: Insecure!



Warning: Broken



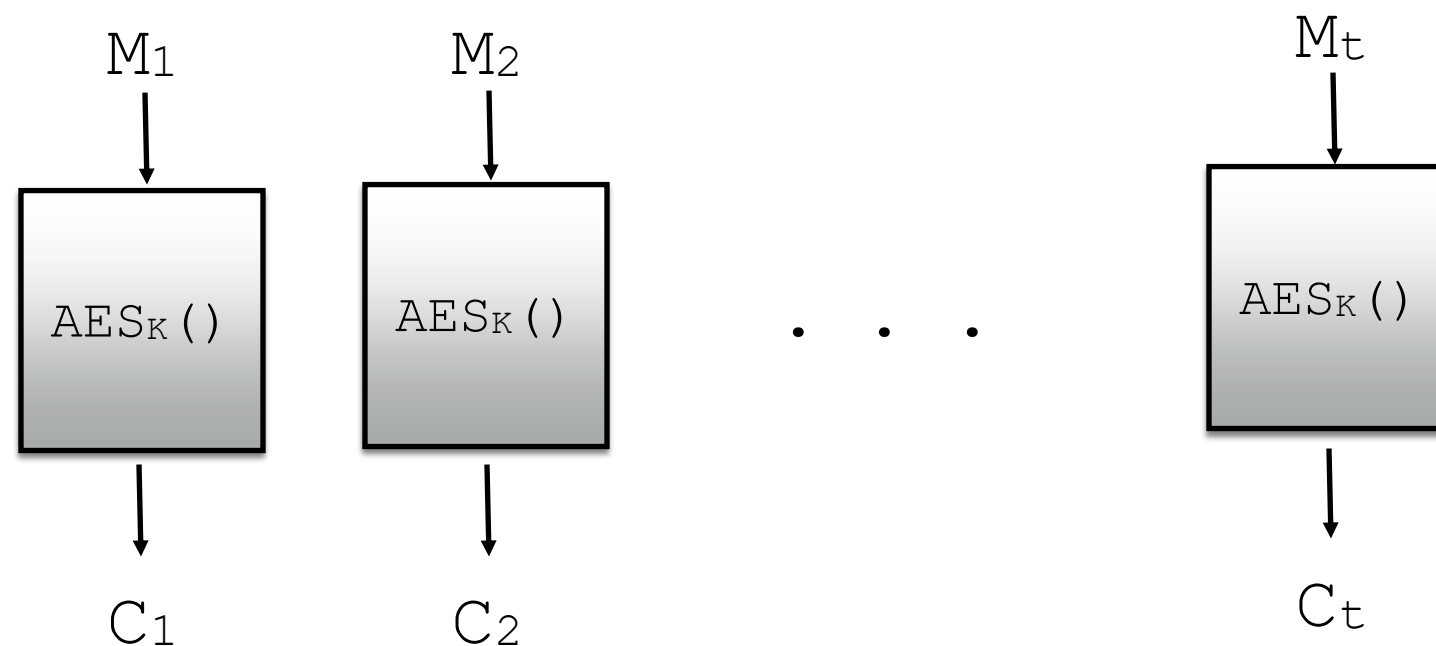
ECB = “Electronic Code Book”

AES-ECB_k(M)

- Split M into blocks M_1, M_2, \dots, M_t
// all blocks except M_t are 16 bytes
- Pad last block, M_t , up to 16 bytes
- For $i=1\dots t$:
 - $C_i \leftarrow \text{AES}_k(M_i)$
- Return C_1, \dots, C_t

Intuitively:

Break message up into 16-byte chunks and encrypt each block with AES.



ECB Mode: Insecure!



Warning: Broken



ECB = “Electronic Code Book”

AES-ECB_k (M)

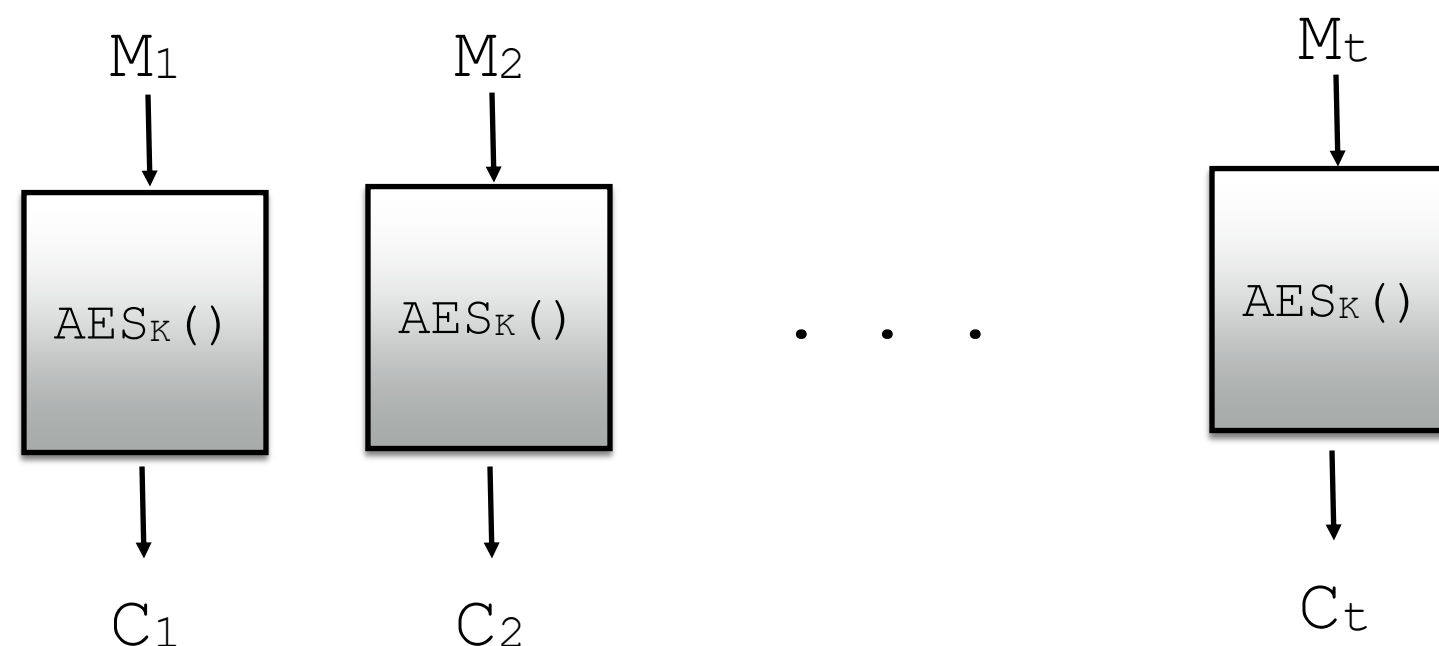
- Split M into blocks M_1, M_2, \dots, M_t
// all blocks except M_t are 16 bytes
- Pad last block, M_t , up to 16 bytes
- For $i=1\dots t$:
 - $C_i \leftarrow \text{AES}_k (M_i)$
- Return C_1, \dots, C_t

Insecure!

Deterministic Encryption:

- The same input (plaintext) always results in the same output (ciphertext).

If any message block repeats, ciphertexts will be identical!



Example: The ECB Penguin



Warning: Broken



- Treat pixel values as one long string & encrypt the string

Plaintext



ECB Ciphertext

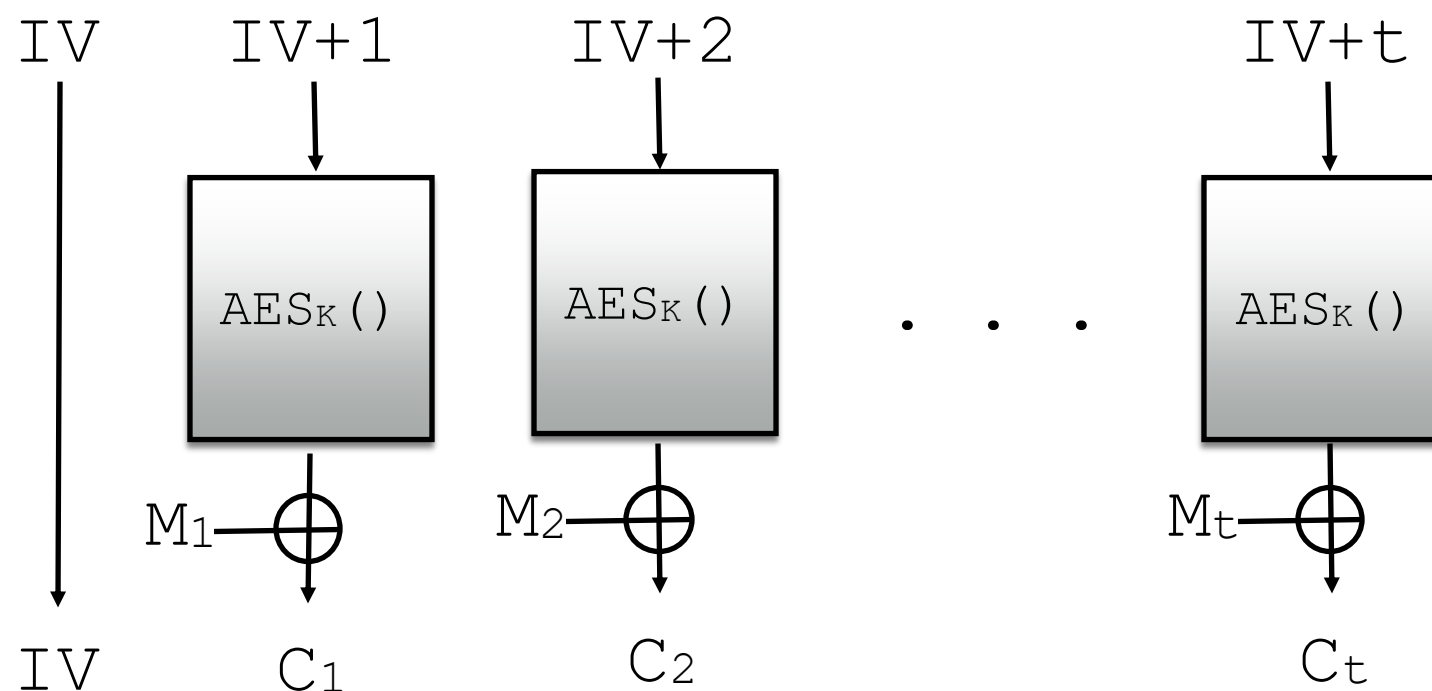


AES-CTR Mode: Secure Confidentiality

- CTR = “Counter Mode”
- Idea: Build a nonce-based stream cipher from AES

AES-CTR_k(IV, M)

- Split M into blocks M_1, M_2, \dots, M_t
// all blocks except M_t are 16 bytes
- For $i=1\dots t$:
 - $C_i \leftarrow M_i \oplus \text{AES}_k(\text{IV}+i)$
- Return $\text{IV}, C_1, \dots, C_t$



IV chosen randomly & transmitted unencrypted.

CTR mode creates “One-Time Pads” for each block, since AES output looks random for different inputs.

Penguin Sanity Check

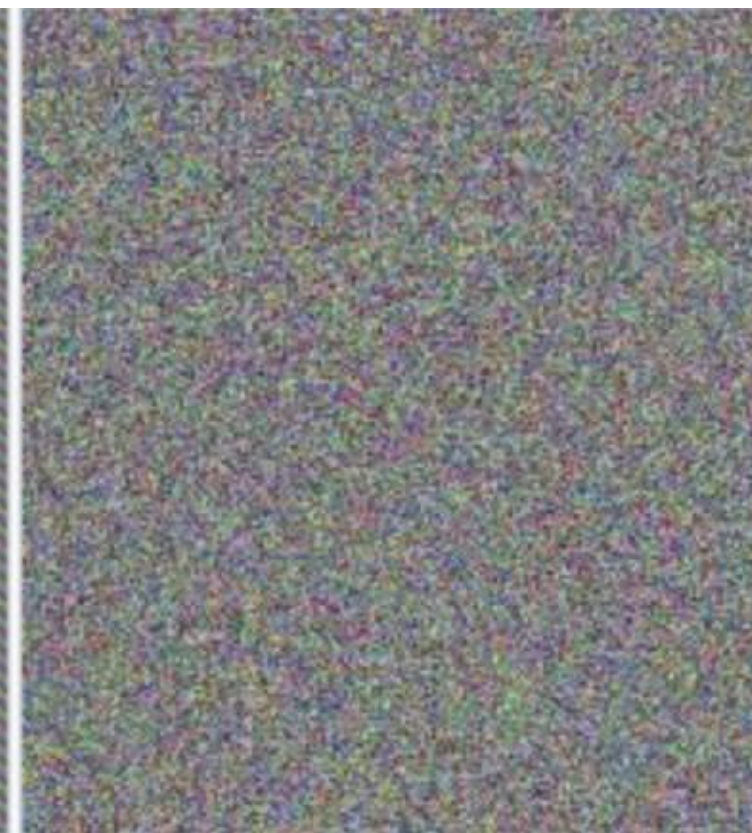
Plaintext



ECB Ciphertext



CTR Ciphertext



Looks random



Encryption Summary

- Security Goal (Confidentiality): given encrypted ciphertexts, the attacker can learn nothing about their plaintext contents
- One-time pads provide theoretically strong security, but are impractical
- Stream ciphers & Block ciphers can achieve practical + secure confidentiality
- Block cipher modes matter for encryption security
 - AES-ECB (naïve block cipher) is INSECURE
 - Modes like AES-CTR and AES-CBC (not discussed) provide secure confidentiality