# Crypto Part 1
# (and Software Defenses Wrap-up)

CMSC 23200, Winter 2024, Lecture 4

Grant Ho and Blase Ur

University of Chicago

# Assignment 2: Logistical Note

For **Problem 4**, your solution *cannot* involve executing any code placed on the stack!

- Currently: configuration error in the VMs that makes Target 4's stack executable.
- However, grading will run your solution against Target 4 with a non-executable stack, so your solution cannot use any shellcode on the stack.
- **Advice:** Implement a return-to-libc attack (as per the instructions)

# 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.

MORE FROM CATALIN CIMPANU

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

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 CICD, 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 and Secure encryption
    - Stream ciphers
    - Message Authentication Codes (MACs)

# What is Cryptography (for CMSC 23200)?

Cryptography develops algorithms that achieve security goals (CIA).

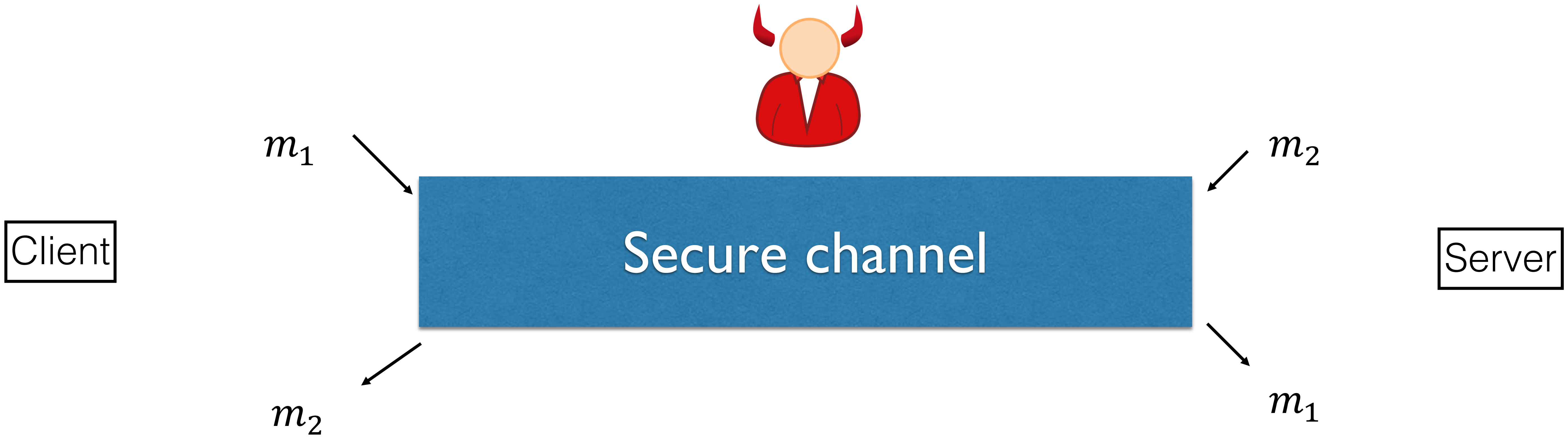
Cryptography involves using math / theory to stop adversaries.

## This Course:

- A brief overview of major crypto concepts and tools
- Cover (some) big “gotchas” in crypto deployments

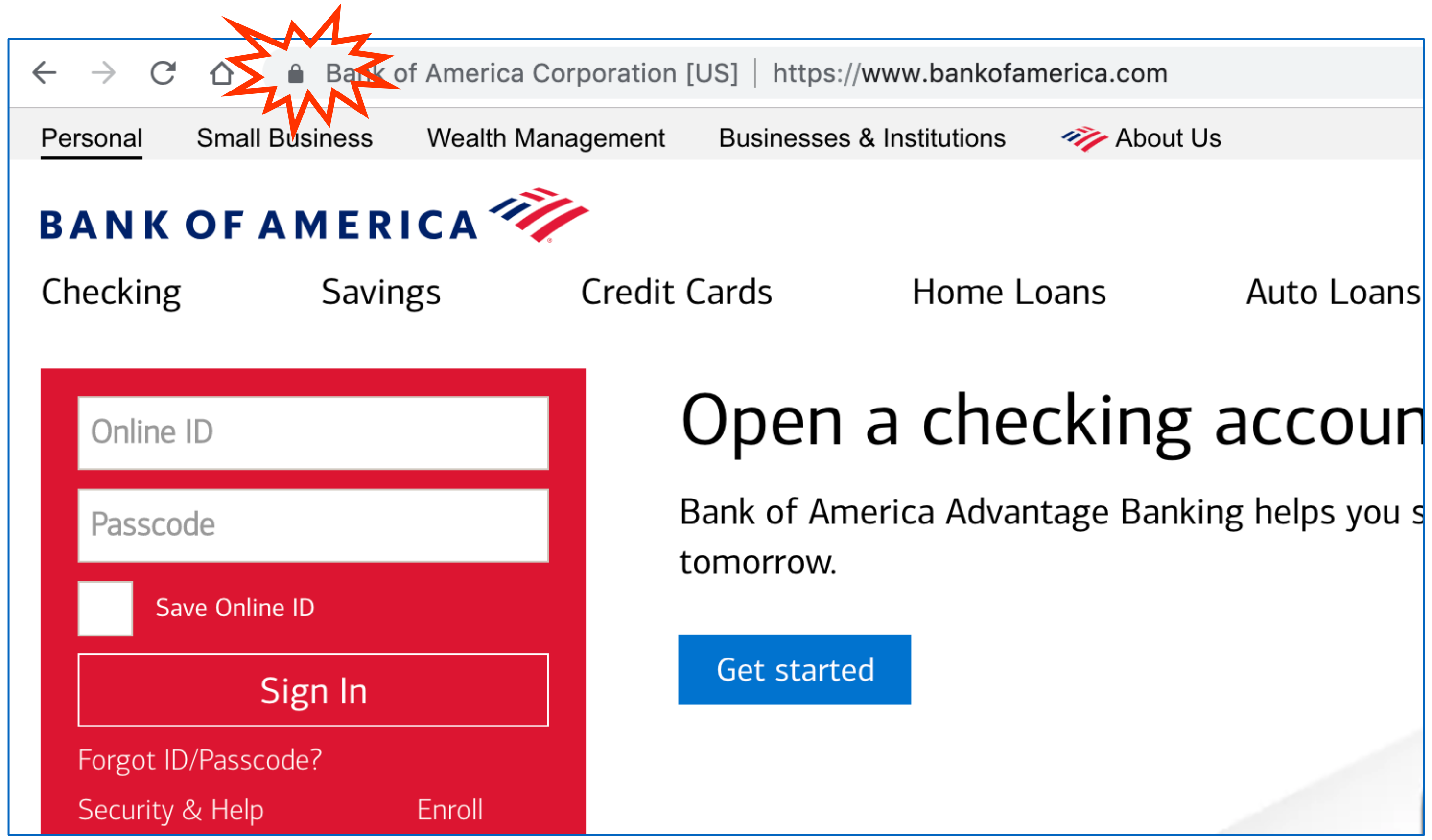
- Not going to cover math, proofs, or many theoretical details.
  Consider taking CS284 (Cryptography)!

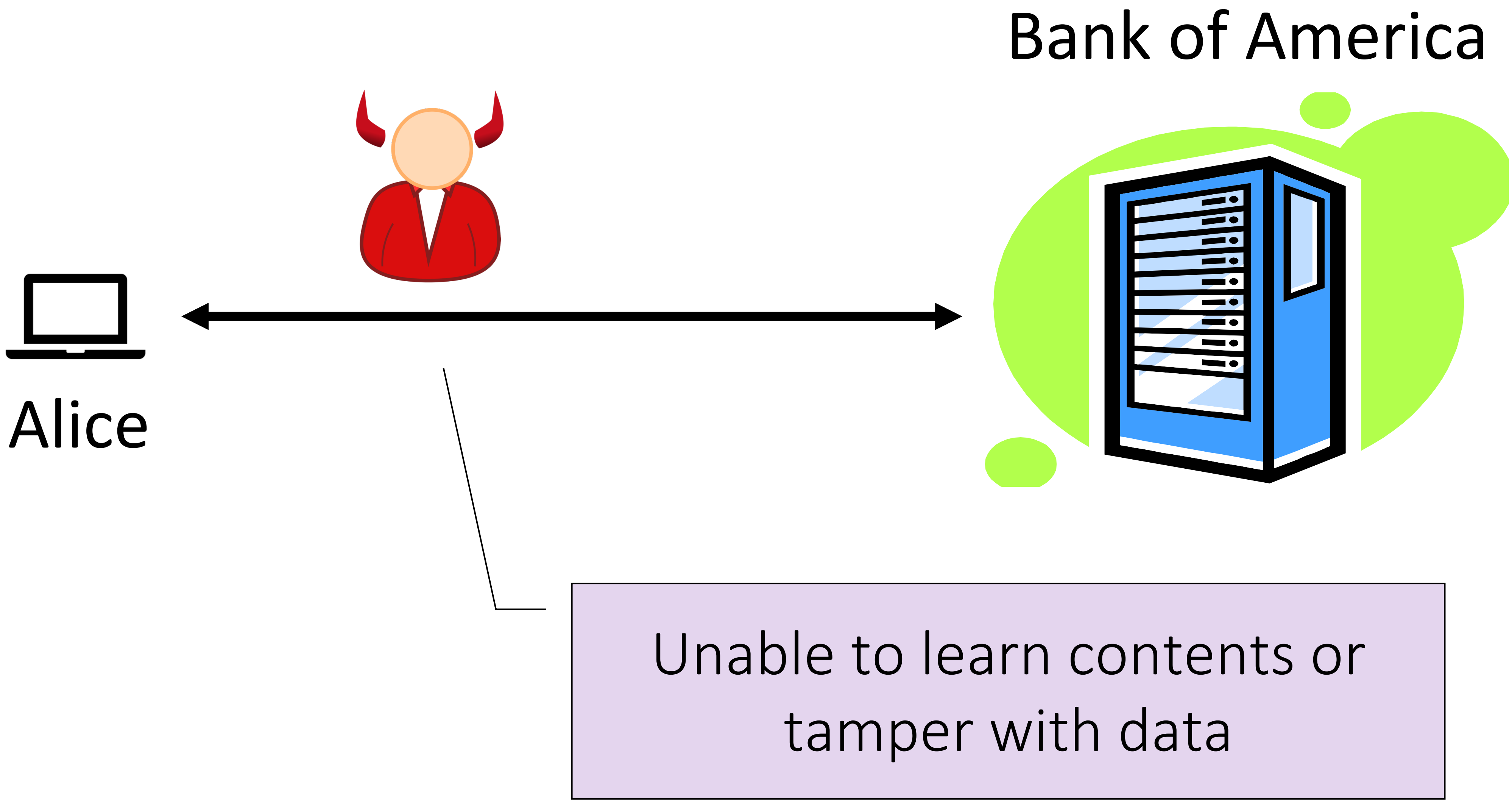
# Common High-Level Goal: Create a Secure Channel



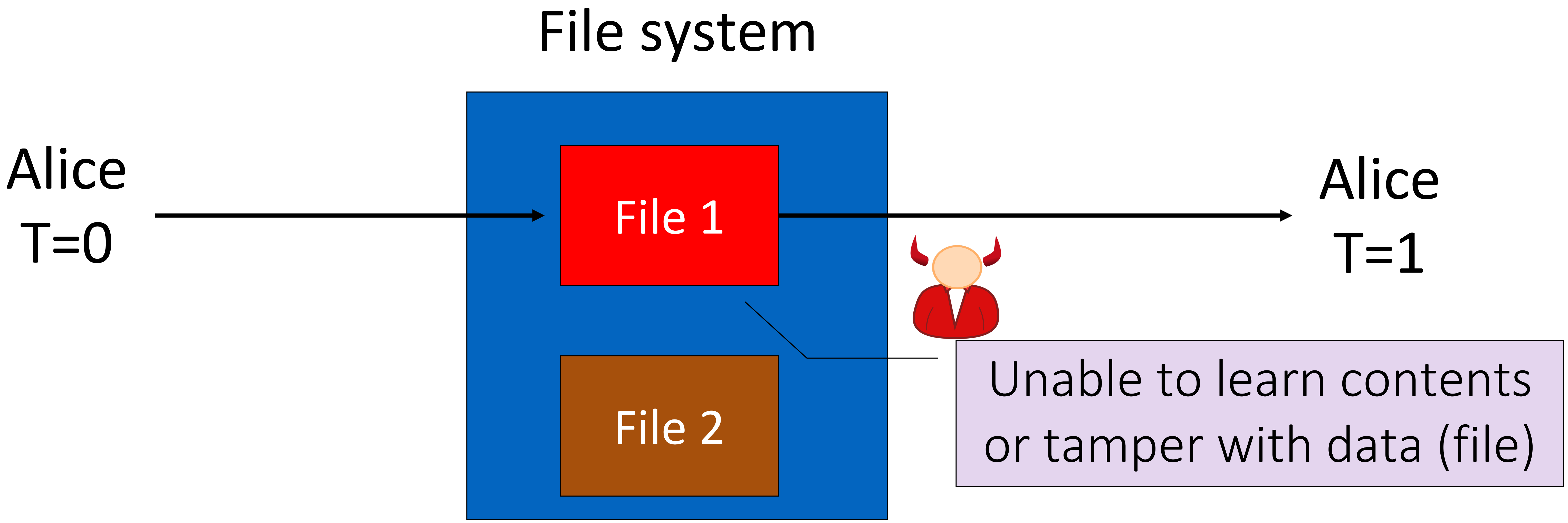

Goal: Attacker does not learn anything about the contents of messages and cannot tamper with their contents.

# 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

| Pre-shared key? \ Security Goal | Confidentiality | Authenticity/Integrity |
| --- | --- | --- |
| Yes (“Symmetric”) | | |
| No (“Asymmetric”) | | |

# Four Cryptography Problems / Tools

| Pre-shared key? \ Security Goal | Confidentiality | Authenticity/Integrity |
|---|---|---|
| Yes (“Symmetric”) | Symmetric Encryption | Message Authentication Code (MAC) |
| No (“Asymmetric”) | | |

# Four Cryptography Problems / Tools

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

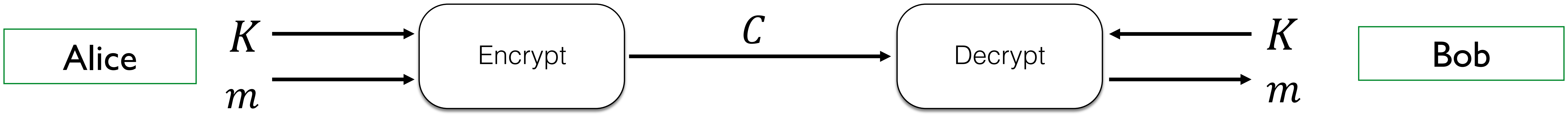
# Outline: Cryptography Part 1

1. Memory Safety Defenses
    - Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
    - Common goals & Threat models
    - Encryption & Basic ciphers
    - One-time pads and Secure encryption
    - Stream ciphers
    - Message Authentication Codes (MACs)

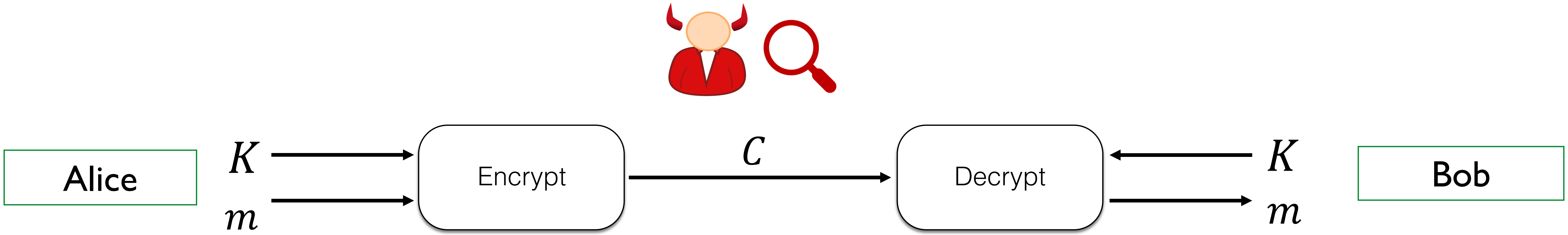
# Ciphers (a.k.a. Symmetric Encryption)

A <u>cipher</u> is a pair of algorithms Encrypt, Decrypt:




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

# Encryption: Providing Confidentiality




**Threat Model: Passive attacker**

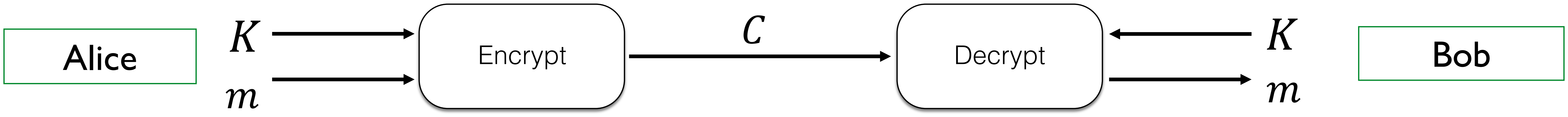
- Adversary see the ciphertexts, but they ***cannot*** modify them in any way
- Attacker's goal: learn something about plaintext messages from ciphertexts

**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.
  - Encrypt(K, m) = c and Decrypt(K, c) = m
- **Confidentiality (Security)**: the ciphertext ***c*** reveals nothing about the message ***m*** (other than the message length)

# 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: `NGGNPXNGQNJA`

# Historical Cipher: Substitution Cipher

Encrypt(K,m): The key K is a permutation $\pi$ on {A,… Z}.
Apply $\pi$ to each character of m to create c

M: ATTACKATDAWN
K: $\pi$
C: ZKKZAMZKYZGT

| x | π(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 |

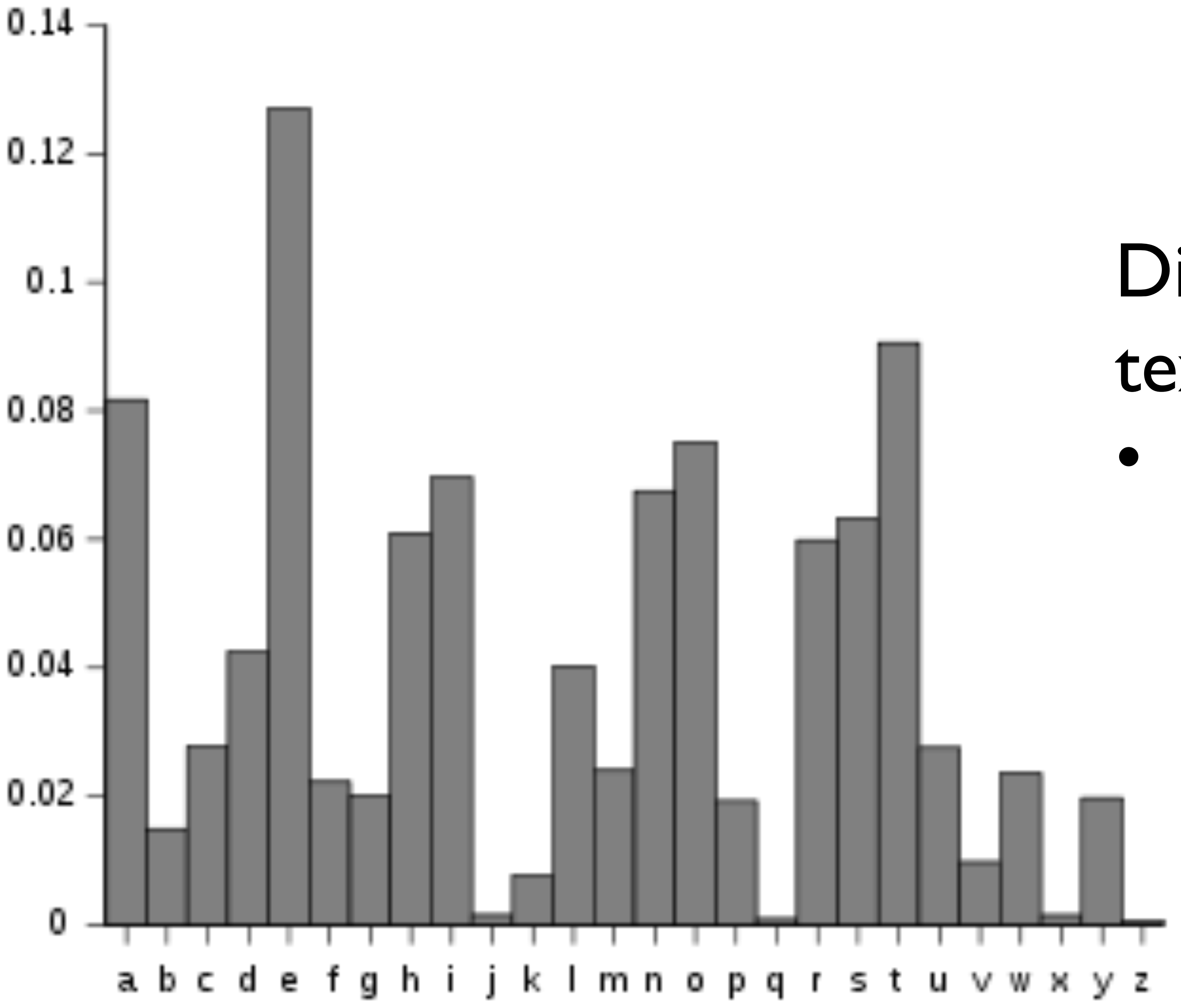
How many keys?

$26! \approx 2^{88}$

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

Q: Is this secure?

# Cryptanalysis of Substitution Cipher




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 and Secure encryption
    - Stream ciphers
    - Message Authentication Codes (MACs)

# Quick recall: Bitwise-XOR operation

We will use bit-wise XOR:

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

<u>Some Properties:</u>

- $X \oplus Y = Y \oplus X$
- $X \oplus X = 000\ldots0$
- $X \oplus Y \oplus X = Y$

# Cipher Example: One-Time Pad (OTP)

Key K: Bitstring of length L

Plaintext M: Bitstring of length L

$\boxed{\text{Encrypt(K,M): Output } K \oplus M}$

$\boxed{\text{Decrypt(K,C): Output } K \oplus C}$

Example:

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

Correctly decrypts because

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

Q: Is the one-time pad secure?

Bigger Q: What does "secure" even mean?

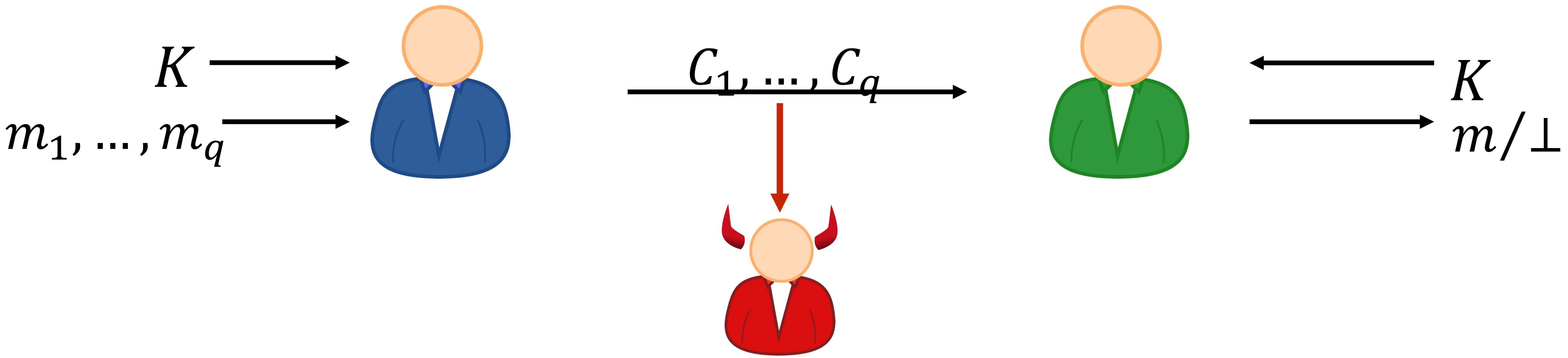
# Evaluating Security of Crypto Algorithms

<u>Kerckhoff's Principle</u>:
Assume the adversary knows your algorithms and implementation. The only thing they don't know is the key.

<u>Example</u>:

- Adversary knows you are running 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

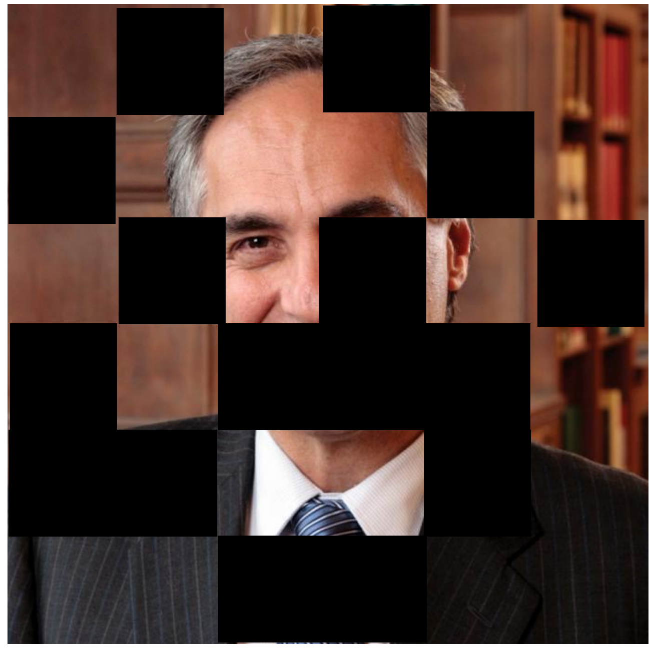



**The adversary sees ciphertexts and attempts to recover some "useful information" about plaintexts.**

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

# 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.

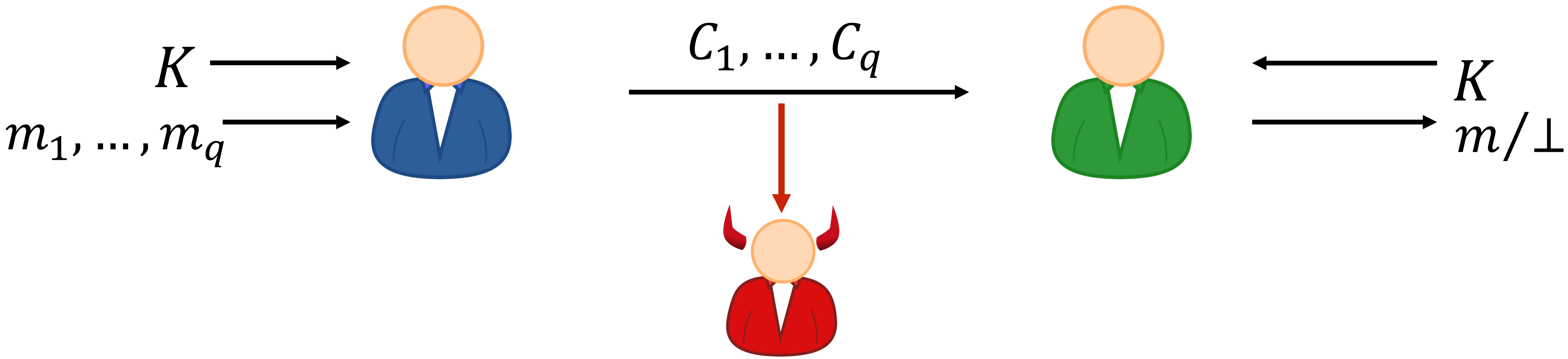
`M = http://site.com?password=`[redacted]

# Secure Encryption Goal

An **attack** is successful as long as it recovers <u>any</u> new info about the plaintext that is useful to the adversary.

Encryption must hide <u>all possible partial information</u> about plaintexts, since what is useful or dangerous is situation-dependent.

# 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.

# Security of the One-Time Pad (OTP)

One-time pad: if an adversary sees **only one** ciphertext under a random key, then <u>any</u> plaintext is equally likely, so they cannot recover any partial information <u>besides the plaintext length</u>.

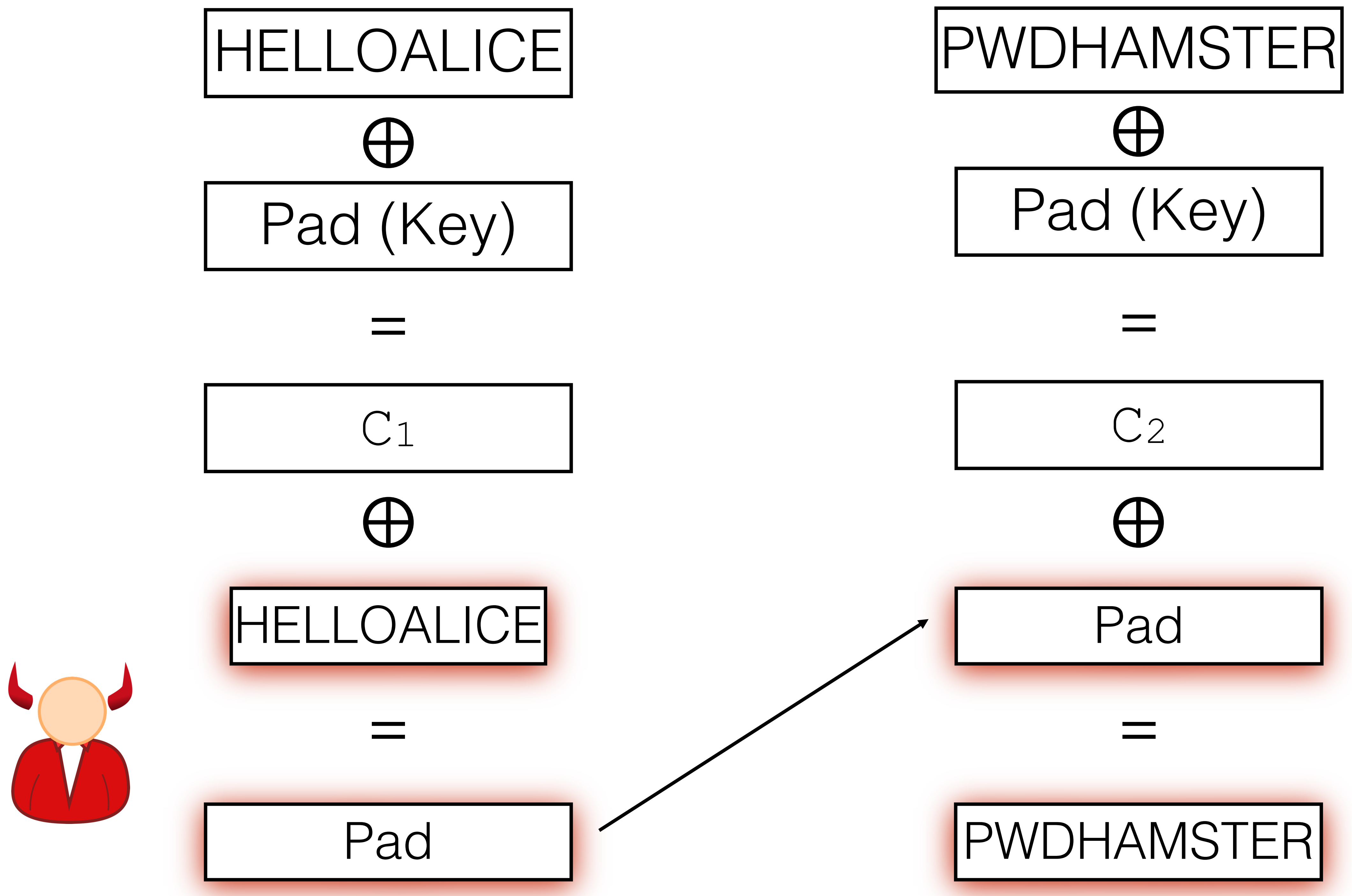
| | |
|---|---|
| Ciphertext observed: | `10111` |
| Possible plaintext: | `00101` |
| ⇒ Possible key: | `10010` |

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]

=  =

$C_1$  $C_2$

$C_1 \oplus C_2$

$= \text{GO\_TO\_THE} \oplus \text{RIVER\_AT\_2}$

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

By definition: OTP needs Key-length ≥ Plaintext-length

- Long message = long key required

In practice:

- Use ***stream cipher***: Encrypt(K,m) = G(K)⊕m
- Use ***nonces*** to encrypt multiple messages (ensures that even if we send same msg twice, the ciphertext is different)

# 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 and Secure encryption
    - Stream ciphers
    - Message Authentication Codes (MACs)

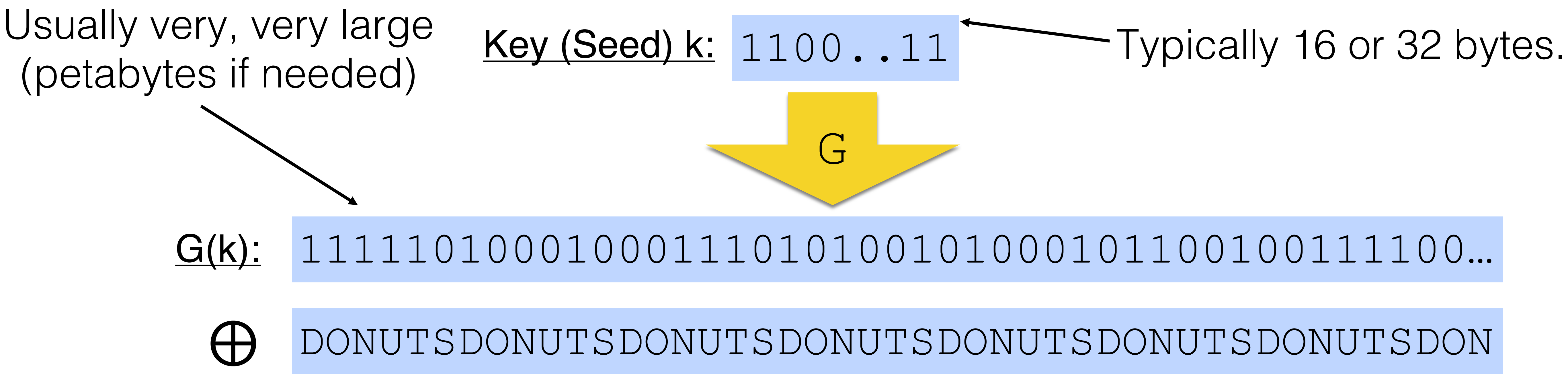
# Tool to address key-length of OTP: Stream Ciphers

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

# Tool to address key-length of OTP: Stream Ciphers

<u>Stream cipher syntax:</u> Algorithm G that takes one (smaller) input and produces a very long bit-string as output.




Use G(key) as the one-time pad.

- Can now encrypt messages much longer than the key.

# 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”.

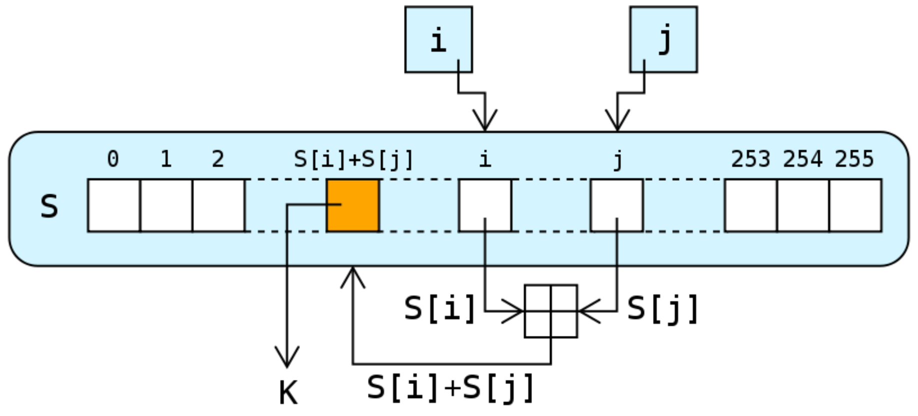
Brute force attack: Given y=G(k), try all possible k and see if you get the string y.

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

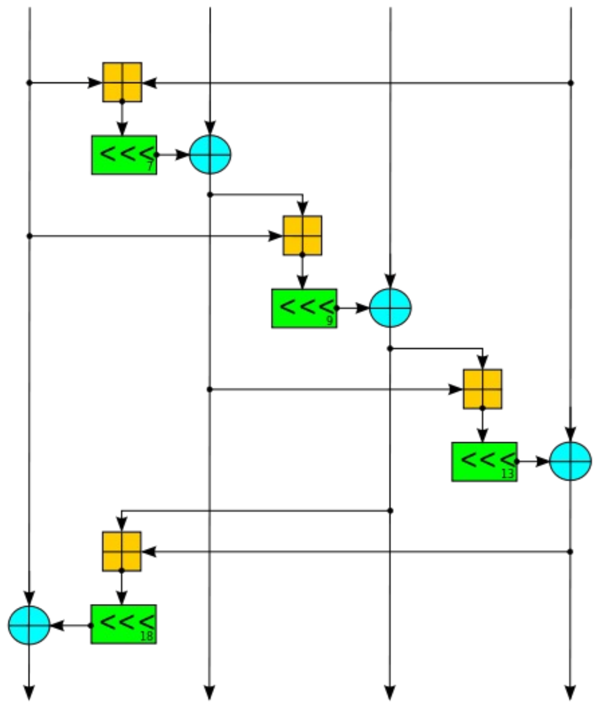
(key length = 256-bits is considered strong now)

# 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?

k

$m_1$

$\oplus$ G(k)

ciphertext

k

$m_2$

$\oplus$ G(k)

ciphertext

...

# 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.

Nonce IV: `1100..11`    Key/Seed k: `1100..11`

G(IV,k): `11111010001000111010100101000101100100111100…`

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

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

# Solution 1: Stream cipher with a nonce

k

IV←0

$m_1$

$\oplus$ G(IV,k)

IV ciphertext

k

IV←IV+1

$m_2$

$\oplus$ G(IV,k)

IV ciphertext

...

- If nonce repeats, then pad repeats

# Example of Pad Re-use: WEP

**Warning: Broken**

<u>IEEE 802.11b WEP:</u> WiFi security standard ’97-‘03

$\boxed{\text{IV}}$

IV is 24-bit wide counter

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

<u>Solutions: (WPA2 replacement)</u>

- 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

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE FORUMS

BIZ & IT —

## Serious flaw in WPA2 protocol lets attackers intercept passwords and much more

KRACK attack is especially bad news for Android and Linux users.

DAN GOODIN - 10/15/2017, 11:37 PM

- Re
- Of

Solutions: (W

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.

- Larger IV sp
- Set IV to combination of packet number, address, etc

# Issues with One-Time Pad

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 and Secure encryption
    - Stream ciphers
    - Message Authentication Codes (MACs)

# Integrity: Message Authentication Codes (MACs)

- Encryption provides confidentiality:
  a *passive* attacker can't learn anything about the data we're storing or using

- Integrity: an (active) attacker cannot tamper with the data in an undetectable manner
  - i.e., allows user to check if the data they received is exactly what was sent or if it has been modified

# Integrity: New Threat Model (Active Attacker)


Threat model: *Active* attacker that can tamper with communication

- Attacker not only sees all ciphertexts, but can also actively **modify** ciphertexts during transmission, **inject** their own data as additional "ciphertexts", **reorder or delete** ciphertexts
- Often known as a Man-in-the-Middle (MITM) attacker

# OTP & Stream Ciphers Do Not Provide Integrity

PAYALICE$1

$\oplus$

Pad

=

C

$\oplus$

000ALICE00

$\oplus$

000DAVID00

=

C’

Decrypt(Pad, C’) = PAYDAVID$1

# Stream ciphers do not give integrity

```
M = please pay ben 20 bucks

C = b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e446a782871c2d
```

```
C’= b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e546a782871c2d

M’  = please pay ben 21 bucks
```

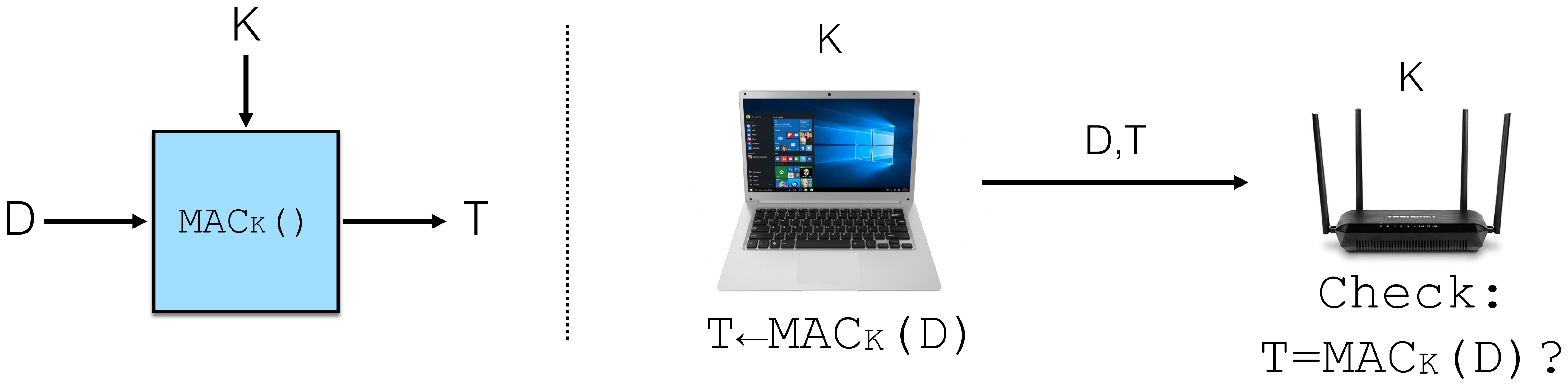
Encryption alone does not provide integrity
(fundamentally not designed to)

# Providing Integrity: Message Authentication Code

**Idea:** Append a special tag to each message that (1) validates the message content (different msg = different tag) and (2) can only be computed if a user knows the secret key K

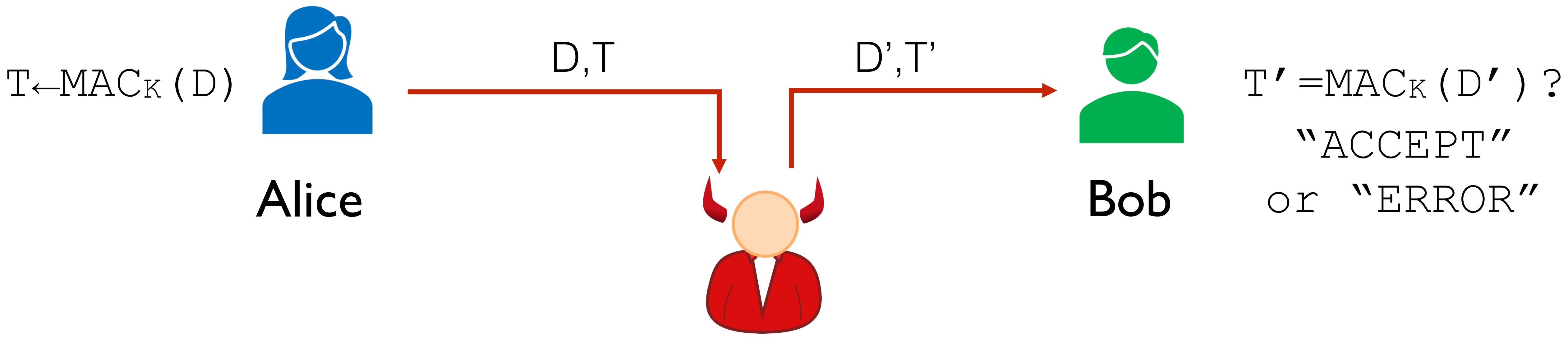
# Providing Integrity: Message Authentication Code

A **message authentication code (MAC)** is an algorithm that takes as input a key and a message, and outputs an "unpredictable" **tag.**




D will usually be a ciphertext, but is often called a "message".

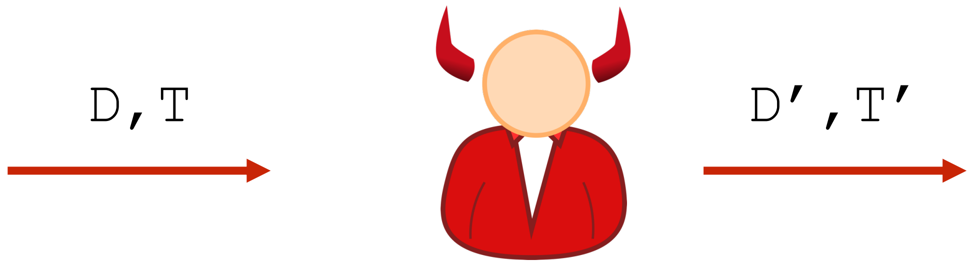
# MAC Security Goal: Unforgeability




MAC satisfies **unforgeability** if it is infeasible for Adversary to fool Bob into accepting `D'` and `T'` as a valid (msg, MAC) pair,
for a `D'` that has not been previously seen

# MAC Security Goal: Unforgeability

```
D = please pay ben 20 bucks

T = 827851dc9cf0f92ddcdc552572ffd8bc
```



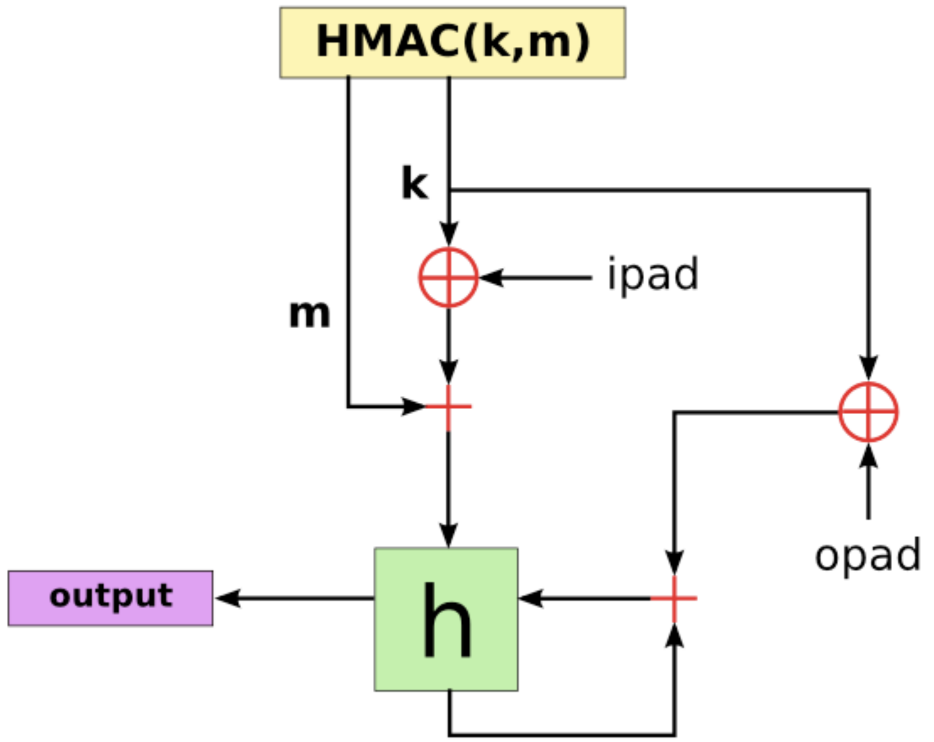

```
D'= please pay ben 21 bucks

T'= baeaf48a891de588ce588f8535ef58b6
```

**Unforgeability:** Attacker cannot create T' for any new D'.

MACs do NOT need to provide any confidentiality (no encryption on this slide)

# MACs In Practice: Use HMAC or Poly1305-AES

- More precisely: Use HMAC-SHA2. More on hashes next lecture




- Other, less-good option: AES-CBC-MAC (bug-prone)

# Authenticated Encryption

Encryption that provides **confidentiality** and **integrity** is called **Authenticated Encryption**.

- Built using a good stream cipher and a MAC.
  - Ex: Salsa20 with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard

The End