Cryptography Part 2

CMSC 23200/33250, Winter 2021, Lecture 8

David Cash & Blase Ur

University of Chicago

Outline

- Message Authentication
- Hash Functions
- Public-Key Encryption
- Digital Signatures

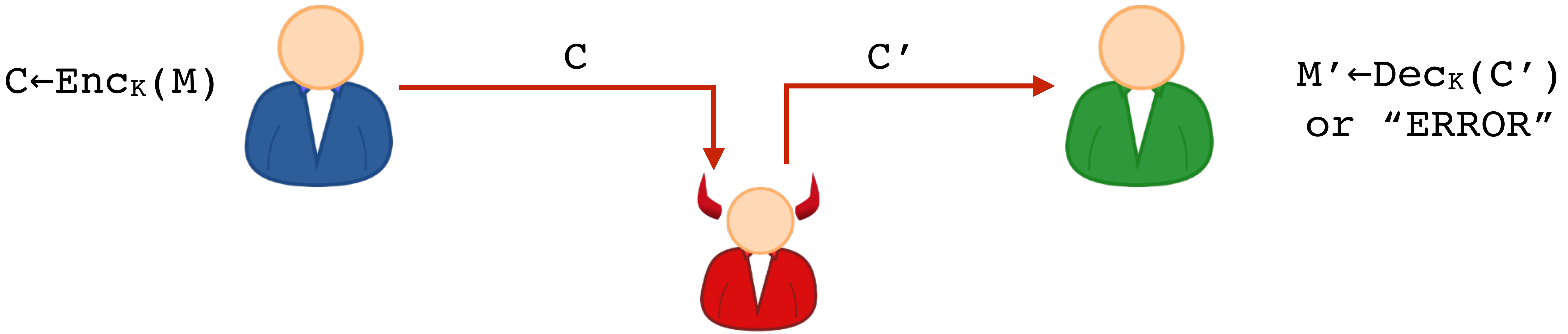
Outline

- **Message Authentication**
- Hash Functions
- Public-Key Encryption
- Digital Signatures

Next Up: Integrity and Authentication

- Authenticity: Guarantee that adversary cannot change or insert ciphertexts
- Achieved with MAC = “Message Authentication Code”

Encryption Integrity: An abstract setting



Encryption satisfies **integrity** if it is infeasible for an adversary to send a new `C'` such that $\mathtt{Dec}_K(\mathtt{C'}) \neq \mathtt{ERROR}$.

AES-CTR does not satisfy integrity

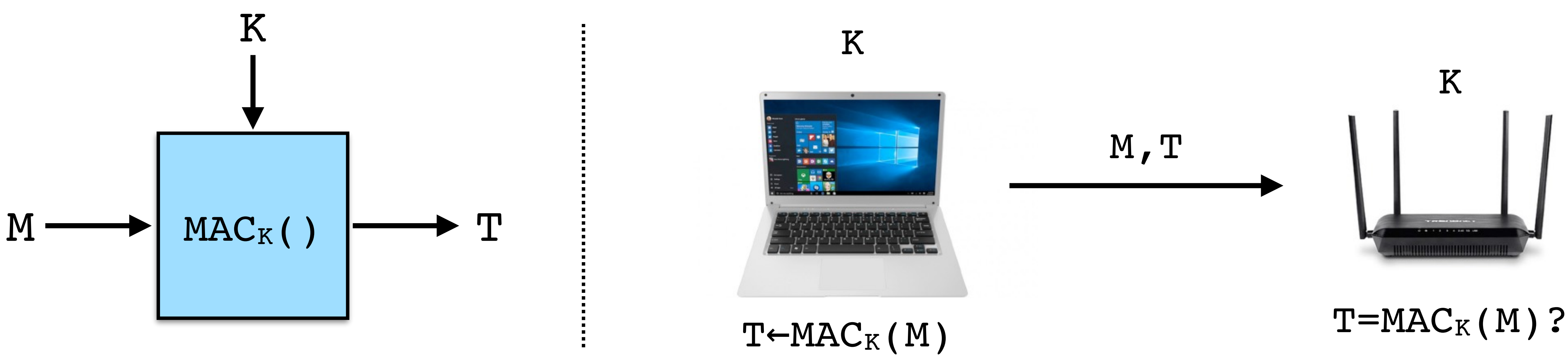
```
M = please pay ben 20 bucks

C = b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e446a782871c2d
```

```
C'= b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e546a782871c2d

M' = please pay ben 21 bucks
```

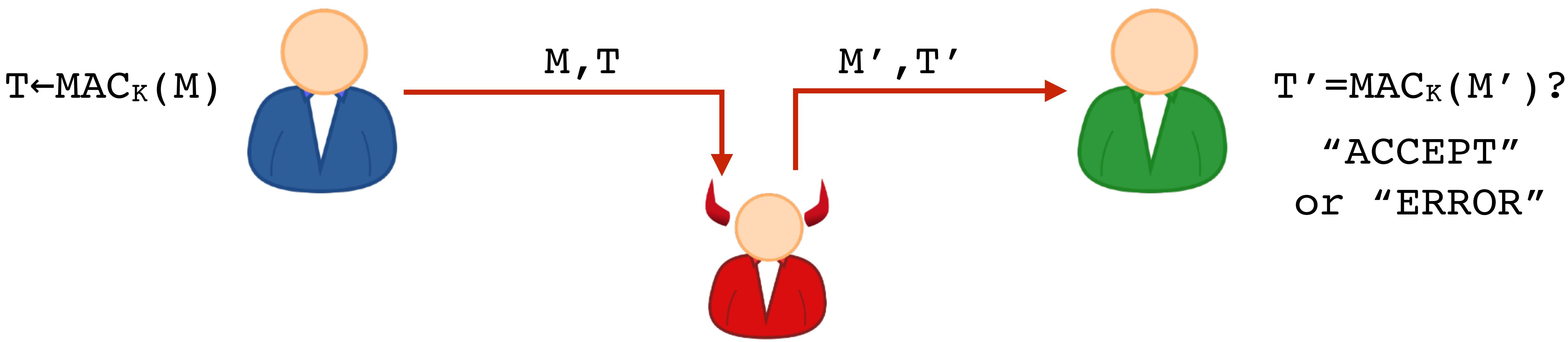
Inherent to stream-cipher approach to encryption.

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.**



MAC Security Goal: Unforgeability



MAC satisfies **unforgeability** if it is unfeasible for Adversary to fool Bob into accepting `M′` not previously sent by Alice.

MAC Security Goal: Unforgeability

Note: No encryption on this slide.

`M = please pay ben 20 bucks`

`T = 827851dc9cf0f92ddcdc552572ffd8bc`

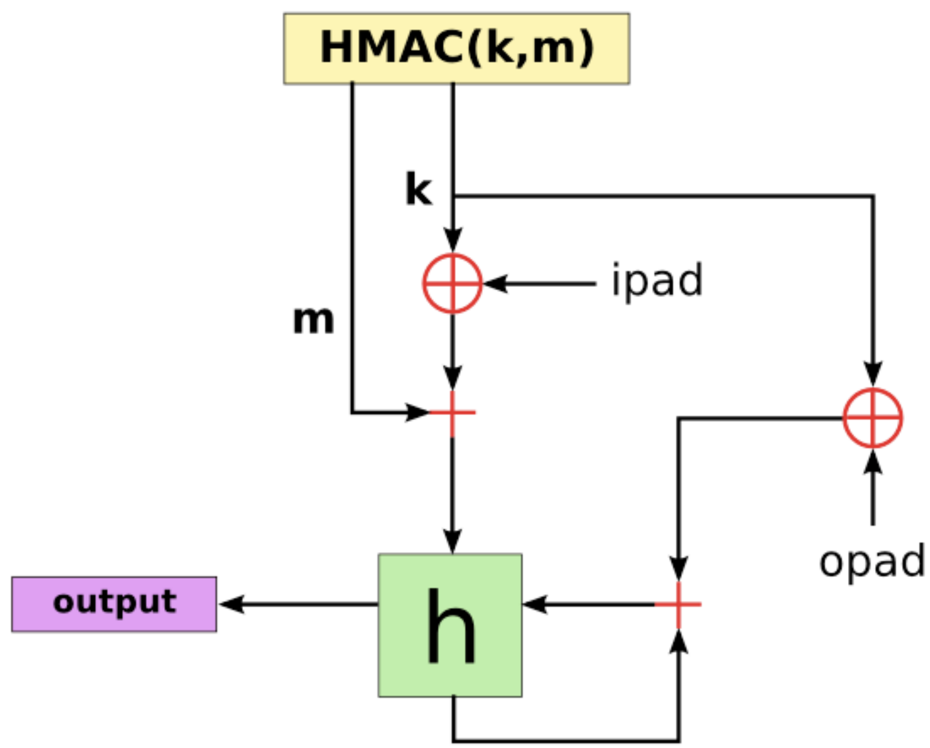
`M’= please pay ben 21 bucks`

`T’= baeaf48a891de588ce588f8535ef58b6`

Should be hard to predict **`T’`** for any new **`M’`**.

MACs In Practice: Use HMAC or Poly1305-AES

- Don't worry about how it works.
- More precisely: Use HMAC-SHA2. More on hashes and MACs later.



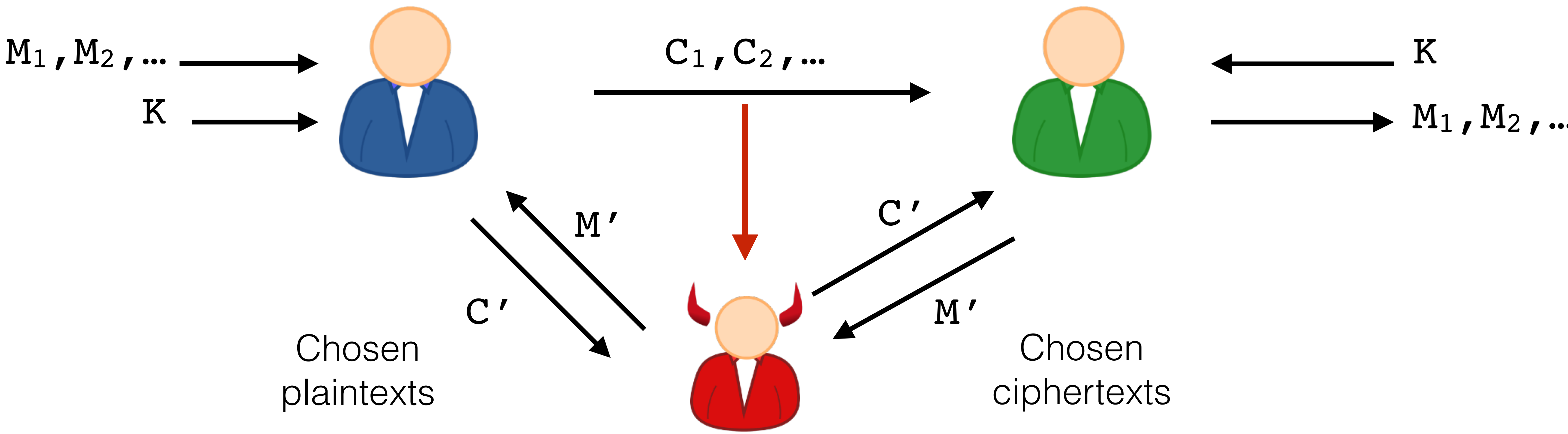
- Other, less-good option: AES-CBC-MAC (bug-prone)

Authenticated Encryption

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

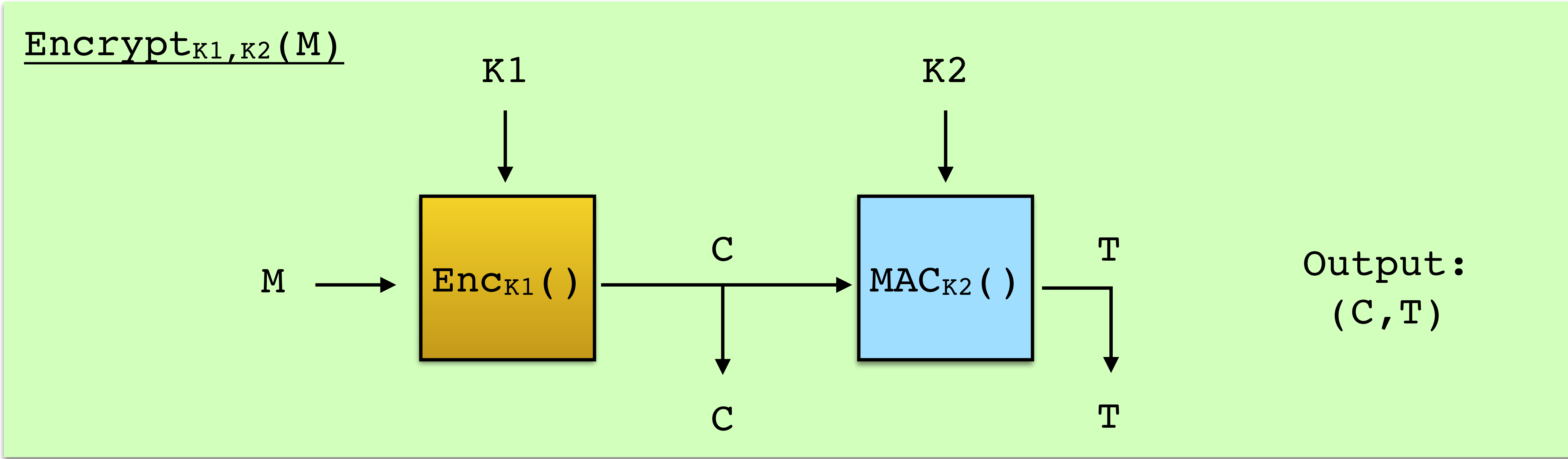
- Built using a good cipher and a MAC.
  - Ex: AES-CTR with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard

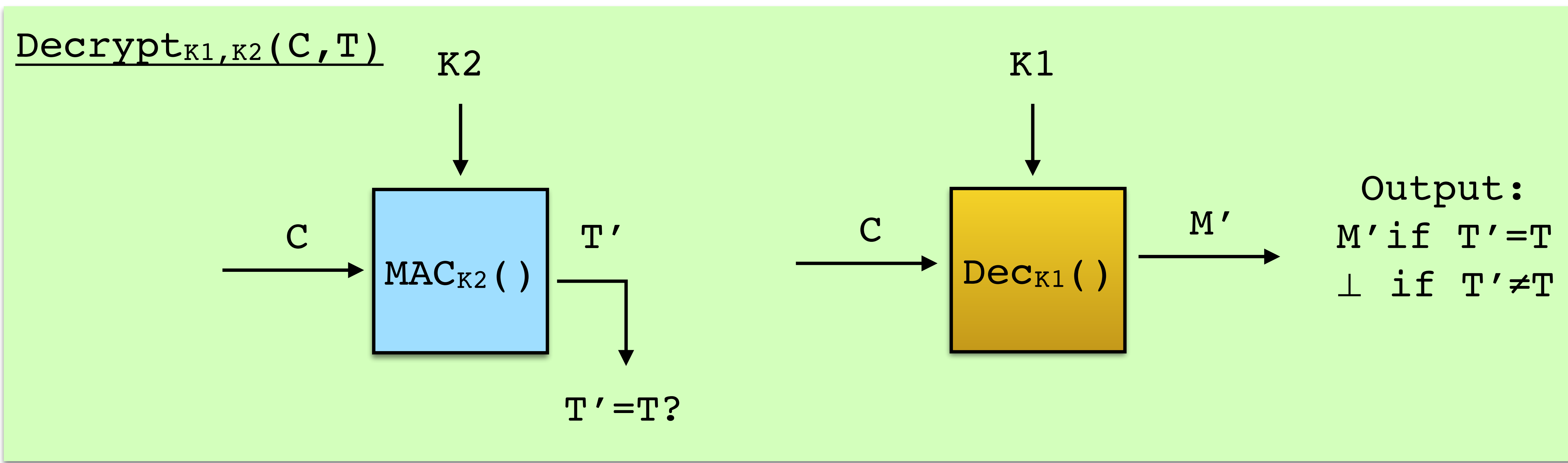
Breaking Encryption: Game with Active Adversaries



Authenticated Encryption Security: Adversary cannot recover any useful information about plaintexts that it didn't form itself OR fool party into accepting some `C'` that wasn't sent.

Building Authenticated Encryption





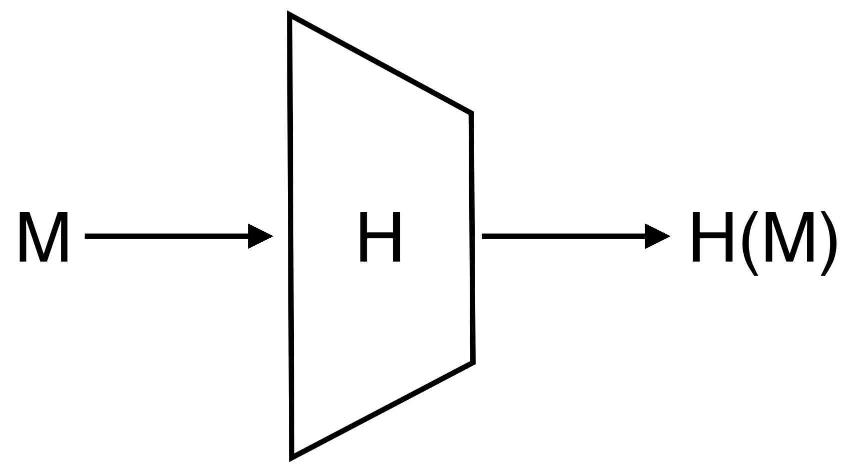
- Summary: MAC the ciphertext, not the message

Outline

- Message Authentication
- **Hash Functions**
- Public-Key Encryption
- Digital Signatures

Next Up: Hash Functions

Definition: A <u>hash function</u> is a deterministic function H that reduces arbitrary strings to fixed-length outputs.

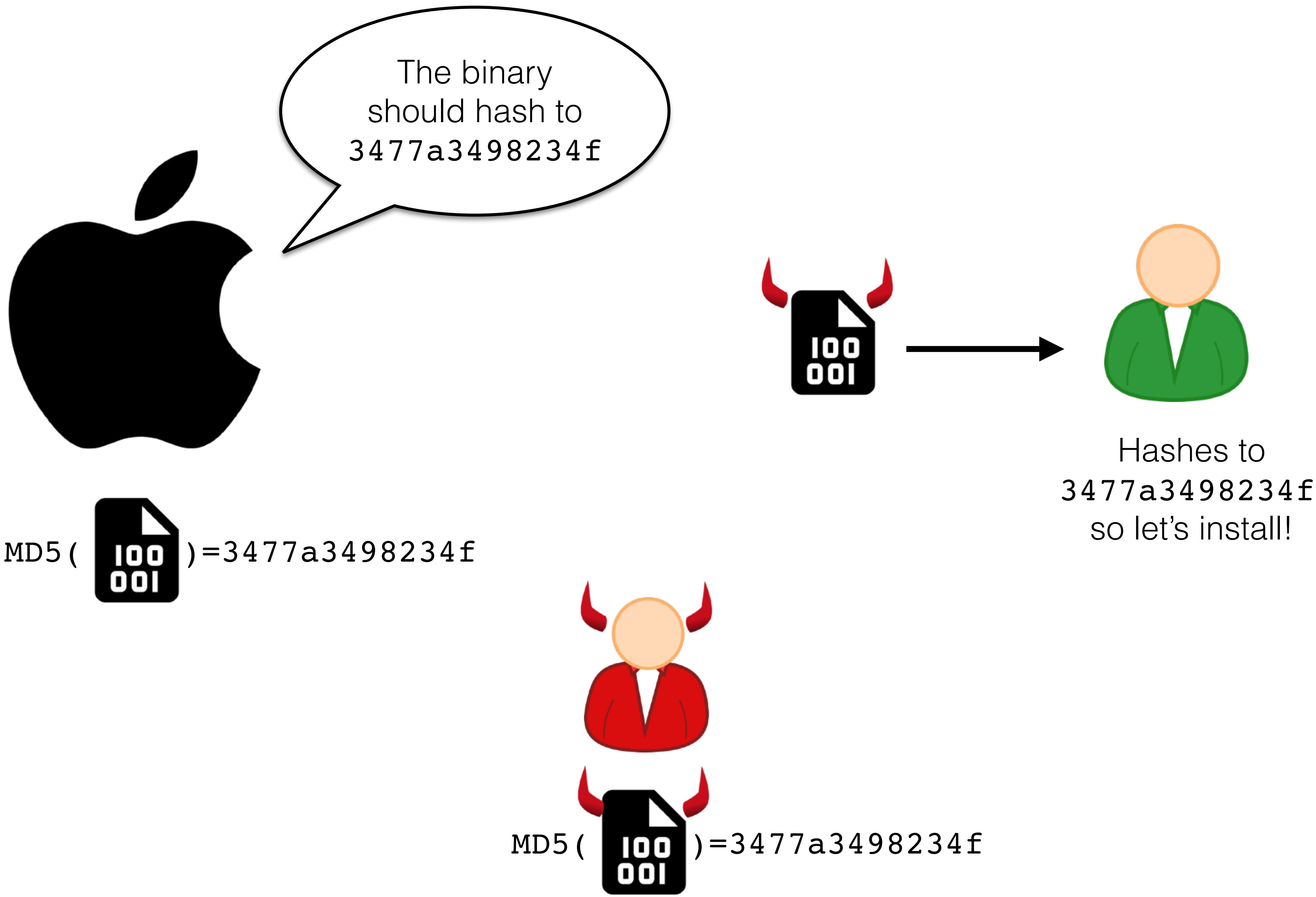


Some security goals:

- collision resistance: can't find M != M' such that H(M) = H(M')
- preimage resistance: given H(M), can't find M
- second-preimage resistance: given H(M), can't find M' s.t. H(M') = H(M)

Note: Very different from hashes used in data structures!

Why are collisions bad?

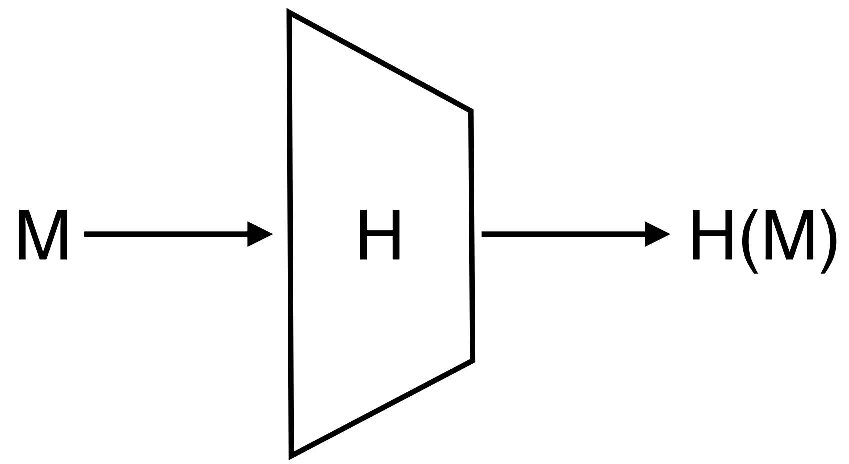


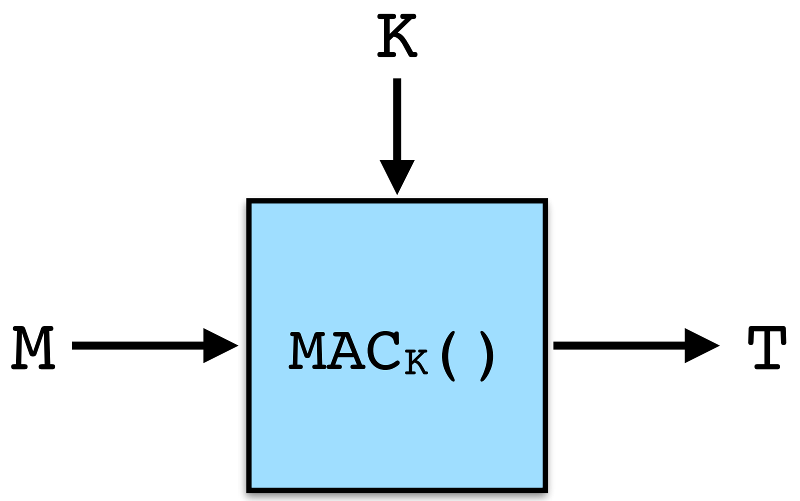
Practical Hash Functions

Name	Year	Output Len (bits)	Broken?
MD5	1993	128	Super-duper broken
SHA-1	1994	160	Yes
SHA-2 (SHA-256)	1999	256	No
SHA-2 (SHA-512)	2009	512	No
SHA-3	2019	>=224	No

Confusion over “SHA” names leads to vulnerabilities.

Hash Functions are not MACs





Both map long inputs to short outputs… But a hash function does not take a key.

Intuition: a MAC is like a hash function, that only the holders of key can evaluate.

MACs from Hash Functions

Goal: Build a secure MAC out of a good hash function.

In Assignment 3: Break this construction!

Construction: $MAC(K, M) = H(K \| M)$ **Warning: Broken**

- Totally insecure if H = MD5, SHA1, SHA-256, SHA-512
- Is secure with SHA-3 (but don't do it)

Construction: $MAC(K, M) = H(M \| K)$ **Just don't**

Upshot: Use HMAC; It's designed to avoid this and other issues.

Later: Hash functions and certificates

Outline

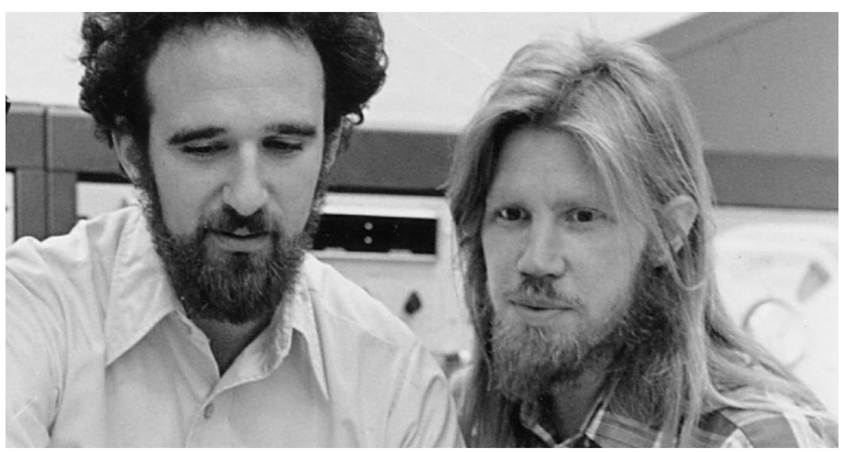
- Message Authentication
- Hash Functions
- **Public-Key Encryption**
- Digital Signatures

Public-Key Encryption

Basic question: If two people are talking in the presence of an eavesdropper, and they don't have pre-shared a key, is there any way they can send private messages?

Public-Key Encryption

Basic question: If two people are talking in the presence of an eavesdropper, and they don't have pre-shared a key, is there any way they can send private messages?

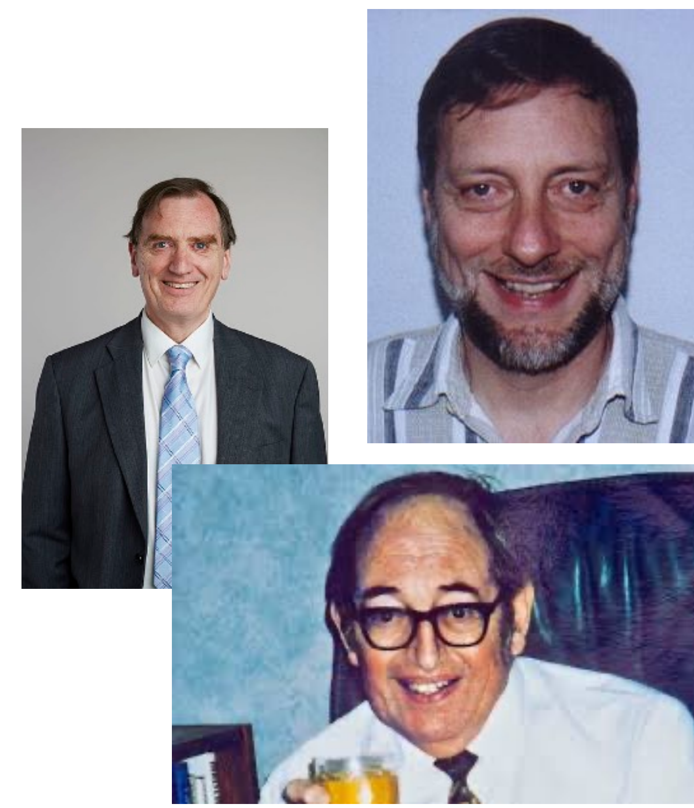


Diffie and Hellman
in 1976: **Yes!**

Turing Award, 2015,
+ Million Dollars

Rivest, Shamir, Adleman
in 1978: **Yes, differently!**

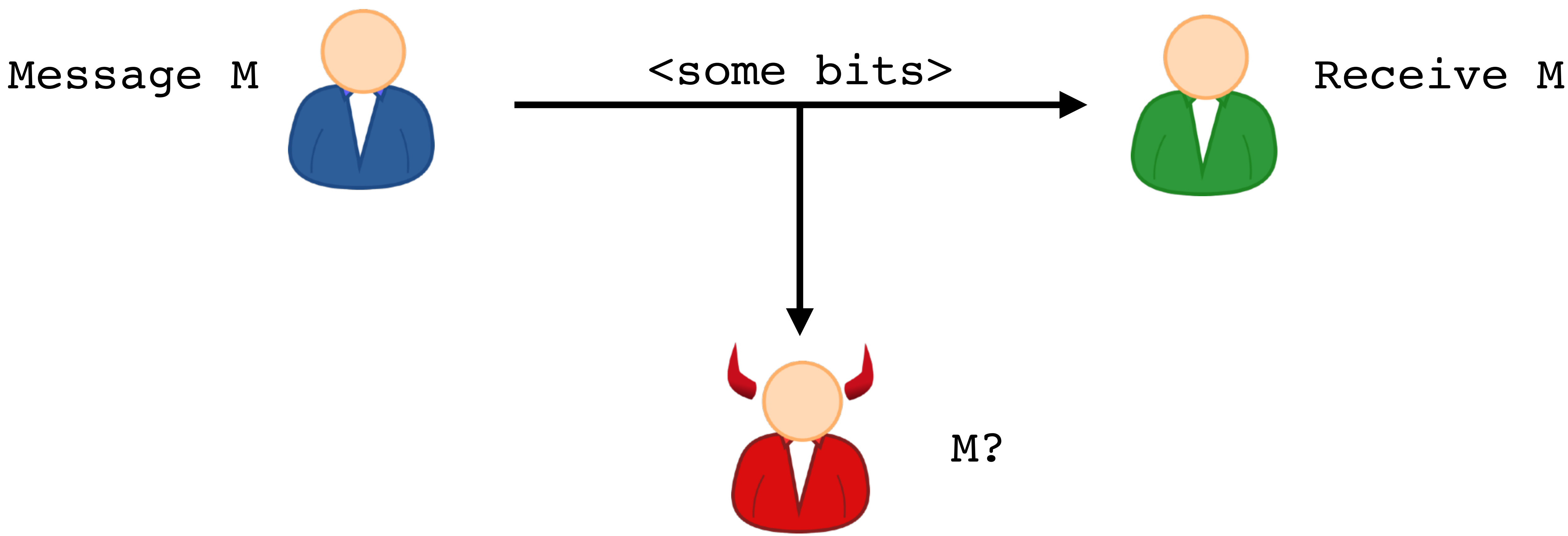
Turing Award, 2002,
+ no money



Cocks, Ellis, Williamson
in 1969, at GCHQ:
Yes…

Public-Key Encryption

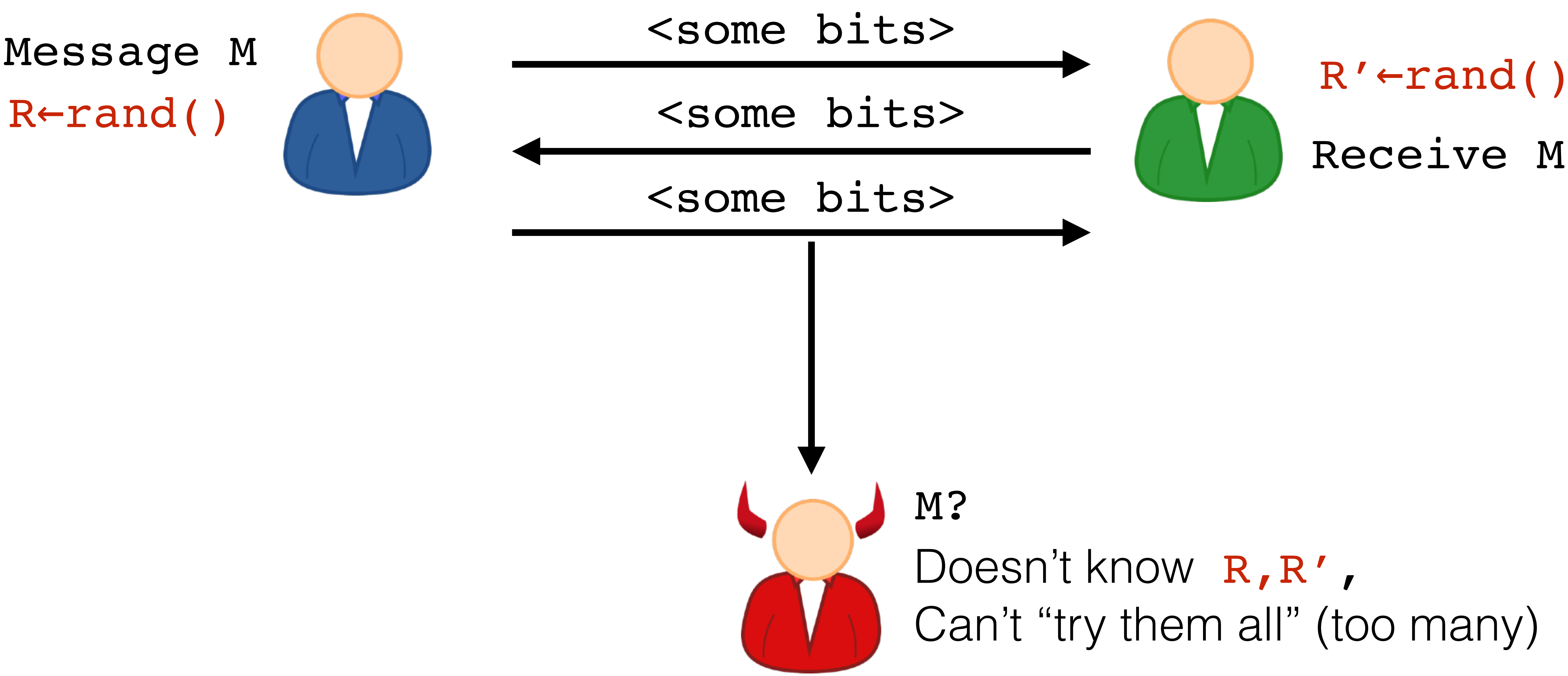
Basic question: If two people are talking in the presence of an eavesdropper, and they don't have pre-shared a key, is there any way they can send private messages?



Formally impossible (in some sense):
No difference between receiver and adversary.

Public-Key Encryption

Basic question: If two people are talking in the presence of an eavesdropper, and they don't have pre-shared a key, is there any way they can send private messages?



Public-Key Encryption

Definition. A public-key encryption scheme consists of three algorithms **Kg**, **Enc**, and **Dec**

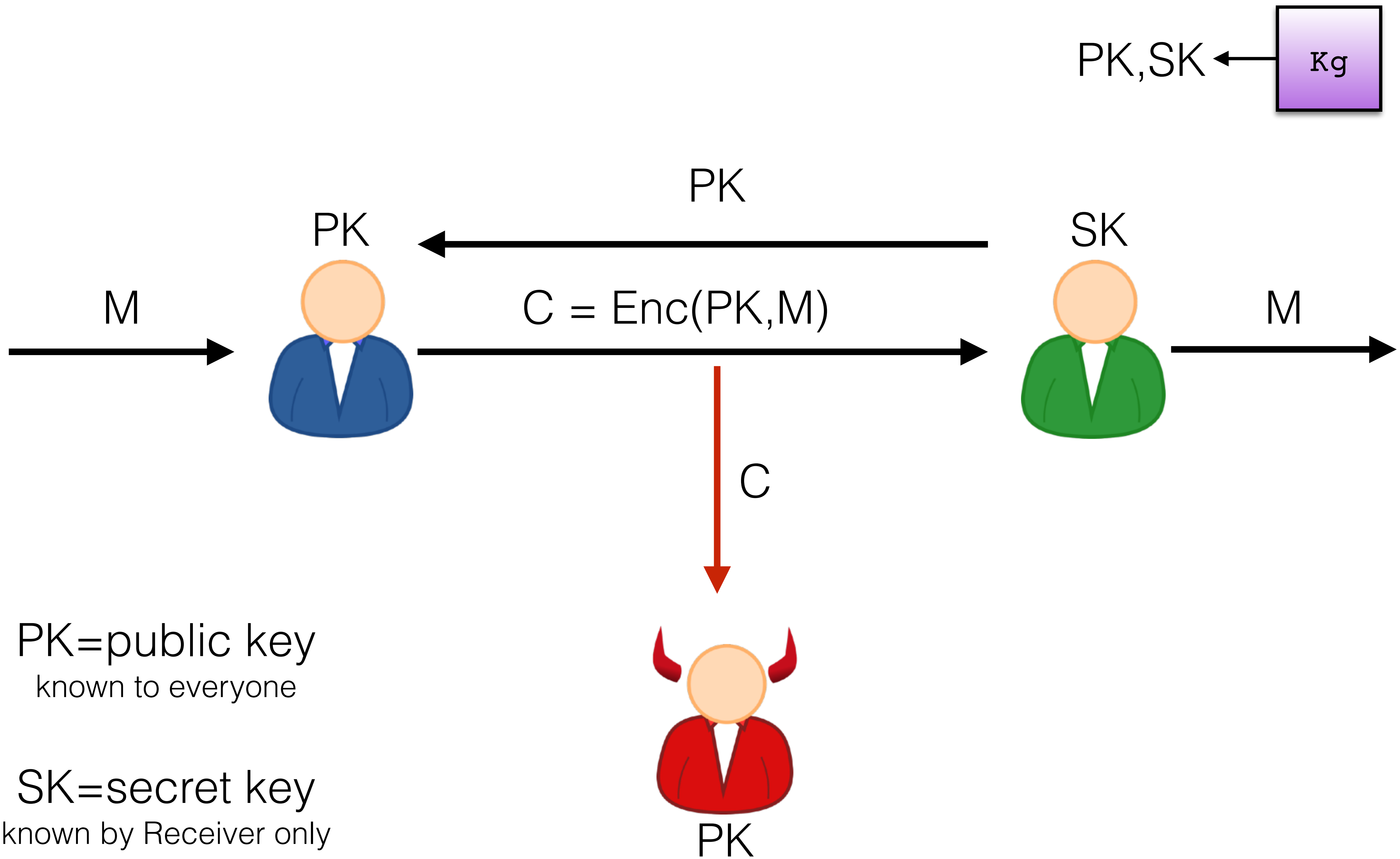
- Key generation algorithm Kg, takes no input and outputs a (random) public-key/secret key pair **(PK,SK)**

- Encryption algorithm Enc, takes input the public key **PK** and the plaintext **M**, outputs ciphertext **C←Enc(PK,M)**

- Decryption algorithm Dec, is such that

$$\mathtt{Dec(SK,Enc(PK,M))=M}$$

Public-Key Encryption in Action



PK=public key
known to everyone

SK=secret key
known by Receiver only

Some RSA Math

Called “2048-bit primes”

RSA setup

p and q be large prime numbers (e.g. around 2^{2048})

$N = pq$

N is called the **modulus**

p=7, q=11 gives N=77

p=17 q=61 gives N=1037

RSA "Trapdoor Function"

$PK = (N, e)$ $\quad SK = (N, d)$ $\quad$ where $\quad N = pq, \quad ed = 1 \bmod \phi(N)$

$$\mathrm{RSA}((N, e), x) = x^e \bmod N$$

$$\mathrm{RSA}^{-1}((N, d), y) = y^d \bmod N$$

Setting up RSA:
- Need two large random primes
- Have to pick e and then find d
- Not covered in 232/332: How this really works.

Never use directly as encryption! **Warning: Broken**

Encrypting with the RSA Trapdoor Function

"Hybrid Encryption":
- Apply RSA to random x
- Hash x to get a symmetric key k
- Encrypted message under k

```
Enc((N,e),M):

1. Pick random x // 0 <= x < N
2. c0←(x^e mod N)
3. k←H(x)
4. c1←SymEnc(k,M) // symmetric enc.
5. Output (c0,c1)
```

```
Dec((N,d), (c0,c1)):

1. x←(c0^d mod N)
2. k←H(x)
3. M←SymDec(k,c1)
4. Output M
```

Do not implement yourself!

Warning: Broken

- Use RSA-OAEP, which uses hash in more complicated way.

Factoring Records and RSA Key Length

- Factoring N allows recovery of secret key
- Challenges posted publicly by RSA Laboratories

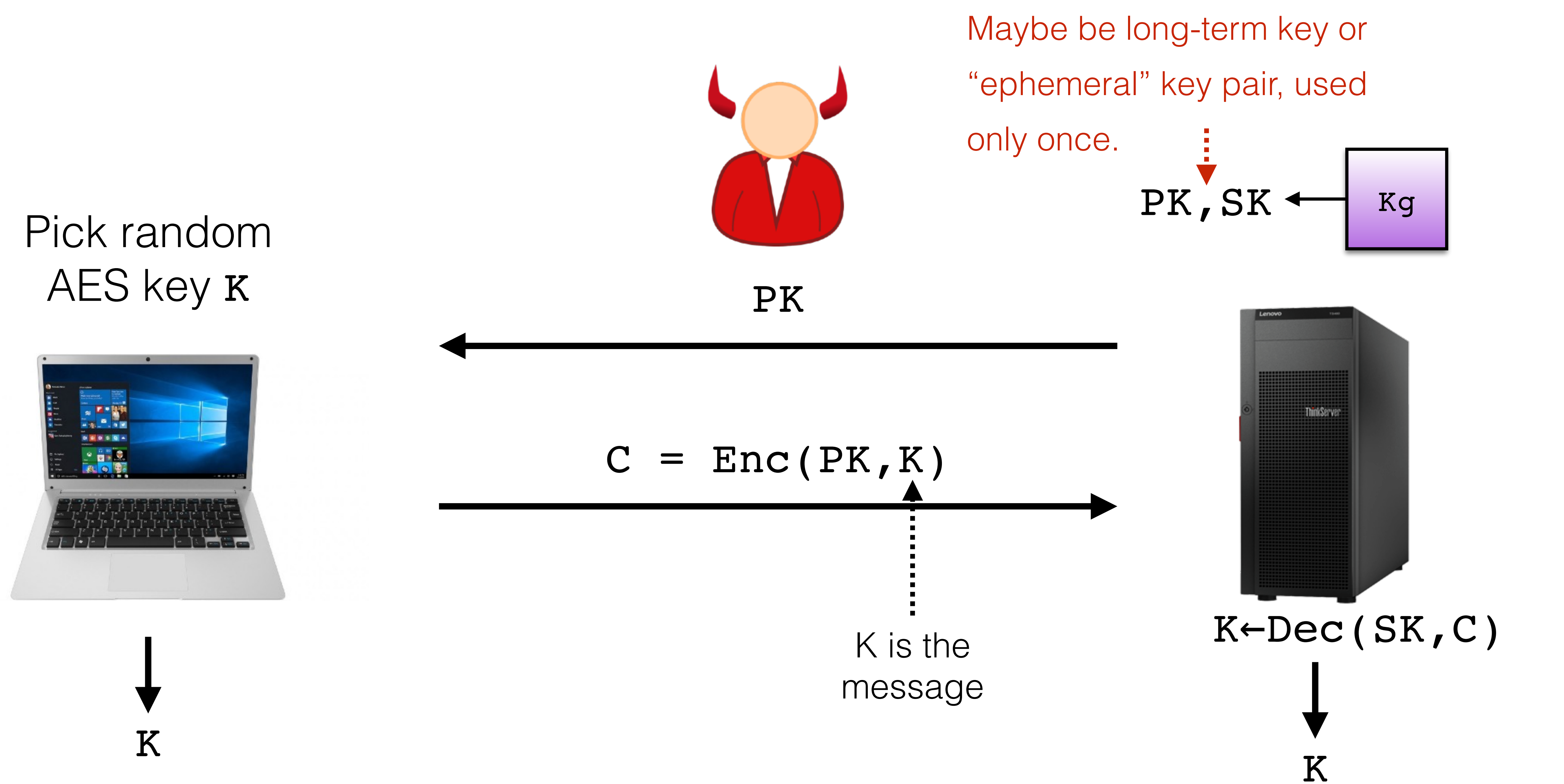
Bit-length of N	Year
400	1993
478	1994
515	1999
768	2009
795	2019

- Recommended bit-length today: 2048
- Note that fast algorithms force such a large key.
  - 512-bit N defeats naive factoring

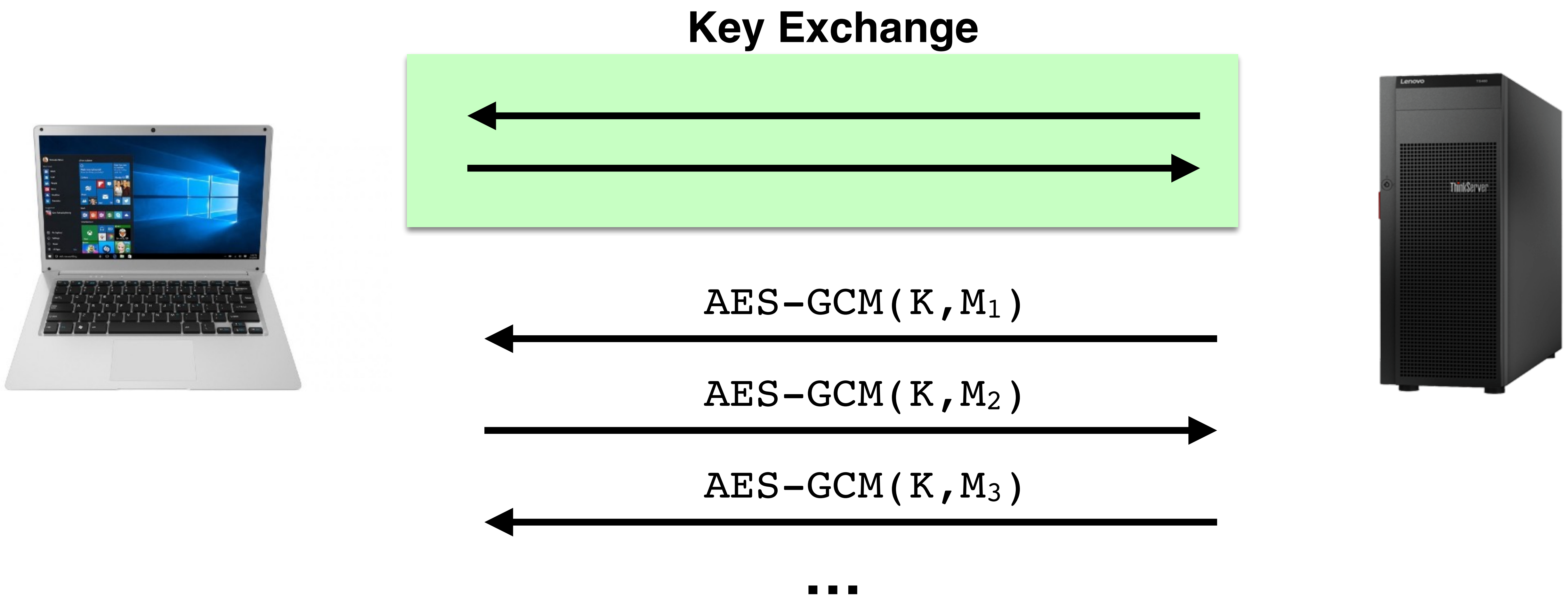
Key Exchange and Hybrid Encryption (TLS next week)

`(Kg, Enc, Dec)` is a public-key encryption scheme.

Goal: Establish secret key `K` to use with Authenticated Encryption.



Key Exchange and Hybrid Encryption



- After up-front cost, bulk encryption is very cheap
- TLS Terminology:
  - “Handshake” = key exchange
  - “Record protocol” = symmetric encryption phase

An alternative approach to key exchange

- They modulus N for RSA is relatively large
  - Mostly important because it slows down encryption/decryption

- Now: A totally different, faster approach based on different math
  - Invented in 1970s, but new ideas have recently made it the standard choice
  - Strictly speaking, not public-key encryption, but can adapted into it if needed

The Setting: Discrete Logarithm Problem

Discrete Logarithm Problem:
Input: Prime $\mathtt{p}$, integers $\mathtt{g}$, $\mathtt{X}$.
Output: integer $\mathtt{r}$ such that $\mathtt{g}^{\mathtt{r}} = \mathtt{X} \bmod \mathtt{p}$.

- Different from factoring: Only one prime.
- Contrast with logarithms with real numbers, which are easy to compute. *Discrete* logarithms appear to be hard to compute
- Largest solved instances: 795-bit prime p (Dec 2019)

Diffie-Hellman Key Exchange

<u>Parameters: (fixed in standards, used by everyone)</u>:
Prime p (1024 bit usually)
Number g (usually 2)

$$(p, g)$$

Diffie-Hellman Key Exchange

<u>Parameters: (fixed in standards, used by everyone)</u>:

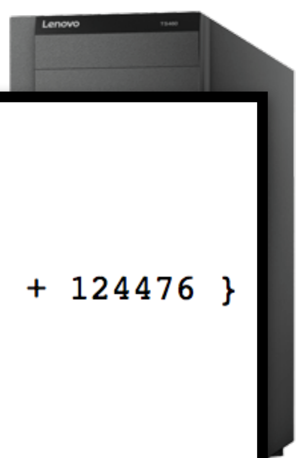
Prime p (1024 bit usually)

Number g (usually 2)

$$(p, g)$$

```
Network Working Group                                        M. Lepinski
Request for Comments: 5114                                       S. Kent
Category: Informational                                BBN Technologies
                                                            January 2008


     Additional Diffie-Hellman Groups for Use with IETF Standards

Status of This Memo

   This memo provides information for
   not specify an Internet standard o
   memo is unlimited.

Abstract

   This document describes eight Diff
   in conjunction with IETF protocols
   communications.  The groups allow
   with a variety of security protoco
   (SSH), Transport Layer Security (T
   (IKE).
```

```
3.  2048-bit MODP Group

   This group is assigned id 14.

   This prime is: 2^2048 - 2^1984 - 1 + 2^64 * { [2^1918 pi] + 124476 }

   Its hexadecimal value is:

      FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
      29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
      EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
      E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
      EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
      C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
      83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
      670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
      E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
      DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
      15728E5A 8AACAA68 FFFFFFFF FFFFFFFF

   The generator is: 2.
```



Diffie-Hellman Key Exchange

<u>Parameters: (fixed in standards, used by everyone)</u>:
Prime p (1024 bit usually)
Number g (usually 2)

$$(p, g)$$

Pick $r_A \in \{1, \ldots, p-1\}$
$X_A \leftarrow g^{r_A} \bmod p$

Pick $r_B \in \{1, \ldots, p-1\}$
$X_B \leftarrow g^{r_B} \bmod p$

$$\xrightarrow{X_A}$$

$$\xleftarrow{X_B}$$

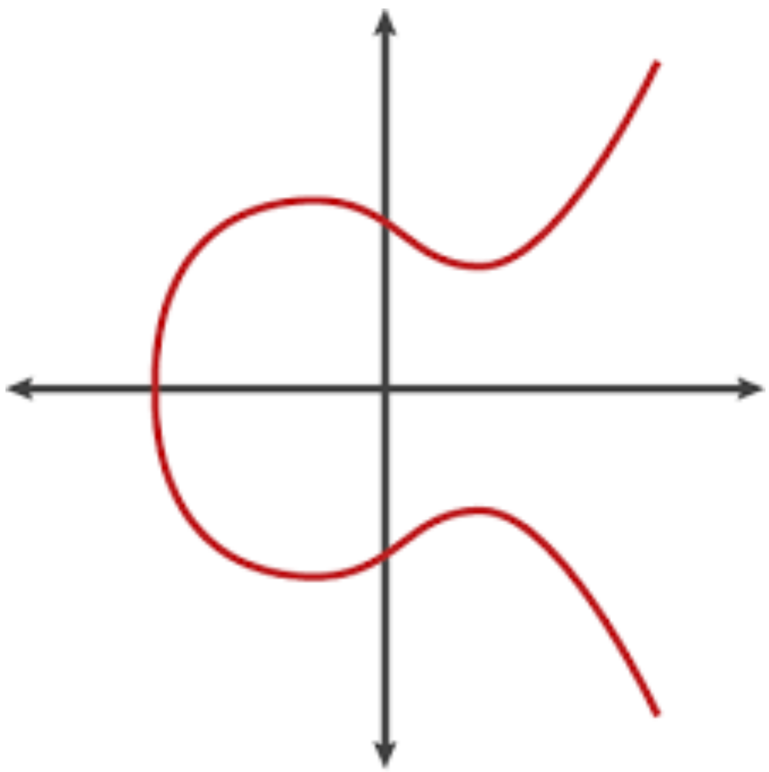
$K \leftarrow X_B^{r_A} \bmod p$

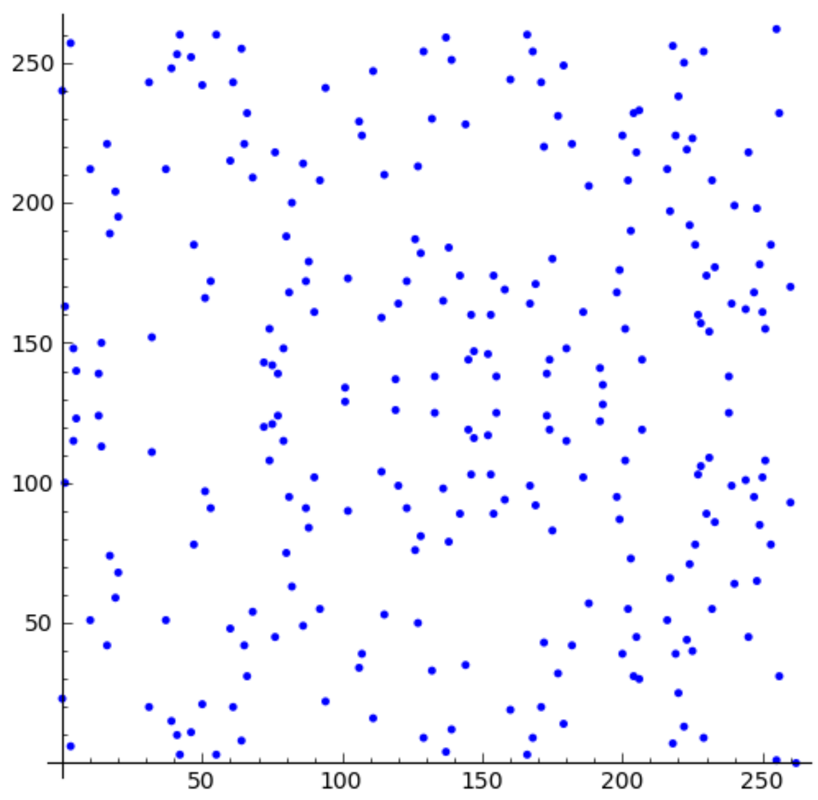
$K \leftarrow X_A^{r_B} \bmod p$

Correctness: $X_B^{r_A} = (g^{r_B})^{r_A} = g^{r_A r_B} = (g^{r_A})^{r_B} = X_A^{r_B} \bmod p$

Modern Key Exchange: *Elliptic Curve* Diffie-Hellman

- Totally different math from RSA
- Advantage: Bandwidth and computation (due to higher security)
  - 256 bit vs 2048-bit messages.





- Used by default in TLS

Public-Key Encryption/Key Exchange Wrap-Up

- RSA-OAEP and Diffie-Hellman (either mod a prime or in an elliptic curve) are unbroken and run fine in TLS/SSH/etc.
- Elliptic-Curve Diffie-Hellman is preferred choice going forward.

Quantum computers will break:
- RSA (any padding)
- Diffie-Hellman

Shor's algorithm, 1994

Peter Shor

- First gen quantum computers will be far from this large
- "Post-quantum" crypto = crypto not known to be broken by quantum computers (i.e. not RSA or DH)
- On-going research on post-quantum cryptography from hard problems on lattices, with first beta deployments in recent years

Outline

- Message Authentication
- Hash Functions
- Public-Key Encryption
- **Digital Signatures**

Crypto Tool: Digital Signatures

Definition. A digital signature scheme consists of three algorithms **`Kg`**, **`Sign`**, and **`Verify`**

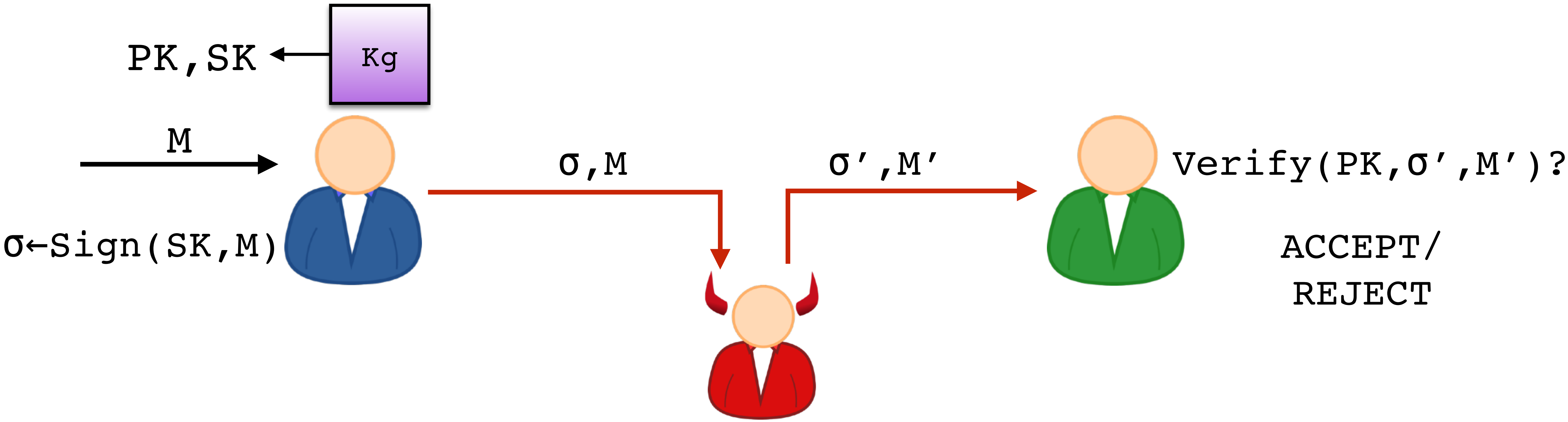
- Key generation algorithm **`Kg`**, takes no input and outputs a (random) public-verification-key/secret-signing key pair `(PK,SK)`

- Signing algorithm **`Sign`**, takes input the secret key `SK` and a message `M`, outputs "signature" $\sigma \leftarrow \texttt{Sign}(\texttt{SK},\texttt{M})$

- Verification algorithm **`Verify`**, takes input the public key `PK`, a message `M`, a signature σ, and outputs `ACCEPT/REJECT`

$$\texttt{Verify}(\texttt{PK},\texttt{M},\sigma)=\texttt{ACCEPT/REJECT}$$

Digital Signature Security Goal: Unforgeability



Scheme satisfies **unforgeability** if it is unfeasible for Adversary (who knows `PK`) to fool Bob into accepting `M′` not previously sent by Alice.

“Plain” RSA with No Encoding

Broken

$PK = (N, e)$ $SK = (N, d)$ where $N = pq$, $ed = 1 \bmod \phi(N)$

$$\mathrm{Sign}((N,d),M) = M^d \bmod N$$

$$\mathrm{Verify}((N,e),M,\sigma) : \sigma^e = M \bmod N?$$

$e = 3$ is common for fast verification; Assume e=3 below.

RSA Signatures with Encoding

$PK = (N, e)$ $SK = (N, d)$ where $N = pq$, $ed = 1 \bmod \phi(N)$

$$\mathrm{Sign}((N,d),M) = \mathrm{encode}(M)^d \bmod N$$

$$\mathrm{Verify}((N,e),M,\sigma) : \sigma^e = \mathrm{encode}(M) \bmod N?$$

encode maps bit strings to numbers between 0 and N

<u>Encoding needs to address:</u>

- Small M or M = perfect cube
- “Malleability”
- “Backwards signing”
- …

```
Encoding must be chosen
with extreme care.
```

Broken

Example RSA Signature: Full Domain Hash

`N`: `n`-byte long integer.
`H`: Hash fcn with `m`-byte output. ← Ex: `SHA-256, m=32`
`k = ceil((n-1)/m)`

```
Sign((N,d),M):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Output σ = Xd mod N
```

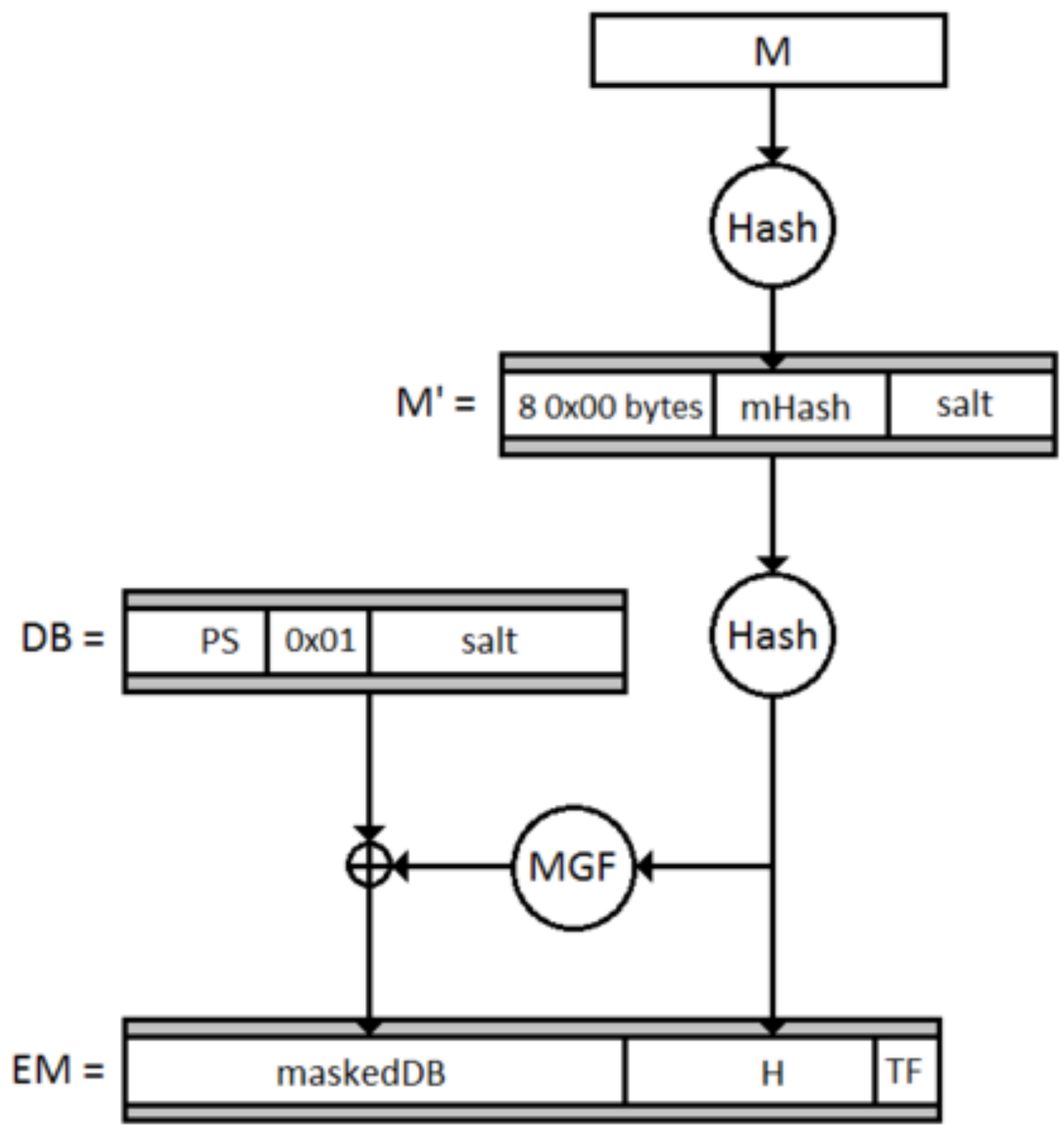
$\sigma = X^d \bmod N$

```
Verify((N,e),M,σ):
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Check if σe = X mod N
```

$\sigma^e = X \bmod N$

Other RSA Padding Schemes: PSS (In TLS 1.3)

- Somewhat complicated
- *Randomized* signing



RSA Signature Summary

- Plain RSA signatures are very broken
- PKCS#1 v.1.5 is widely used, in TLS, and fine if implemented correctly
- Full-Domain Hash and PSS should be preferred
- Don’t roll your own RSA signatures!

Other Practical Signatures: DSA/ECDSA

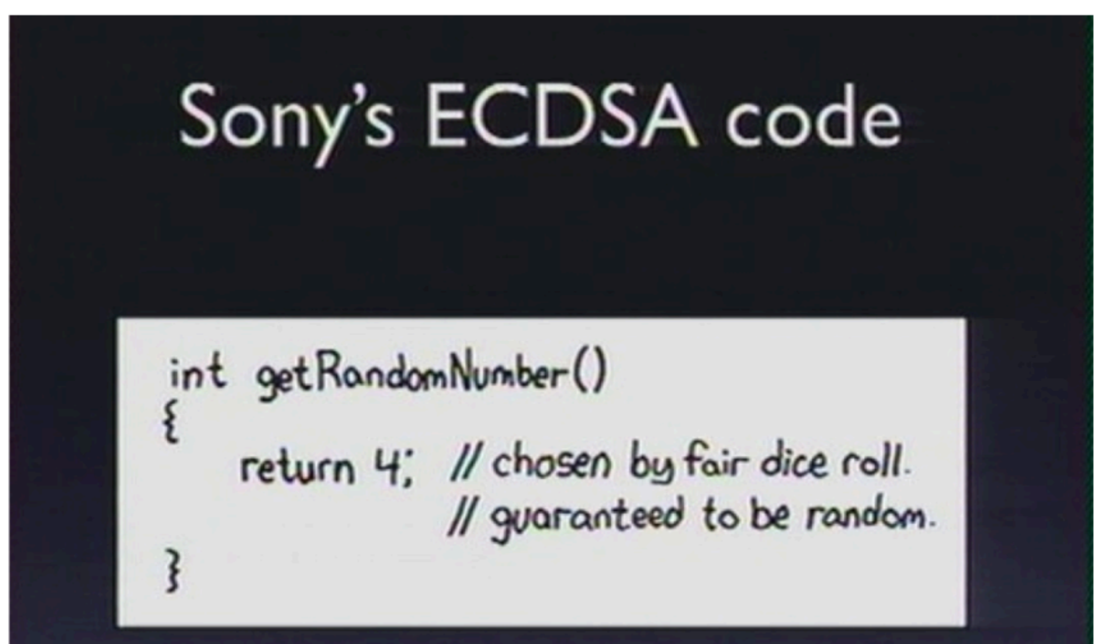
- Based on ideas related to Diffie-Hellman key exchange
- Secure, but even more ripe for implementation errors

—

Hackers obtain PS3 private cryptography key due to epic programming fail? (update)

Sean Hollister
12.29.10

2
Shares



The End