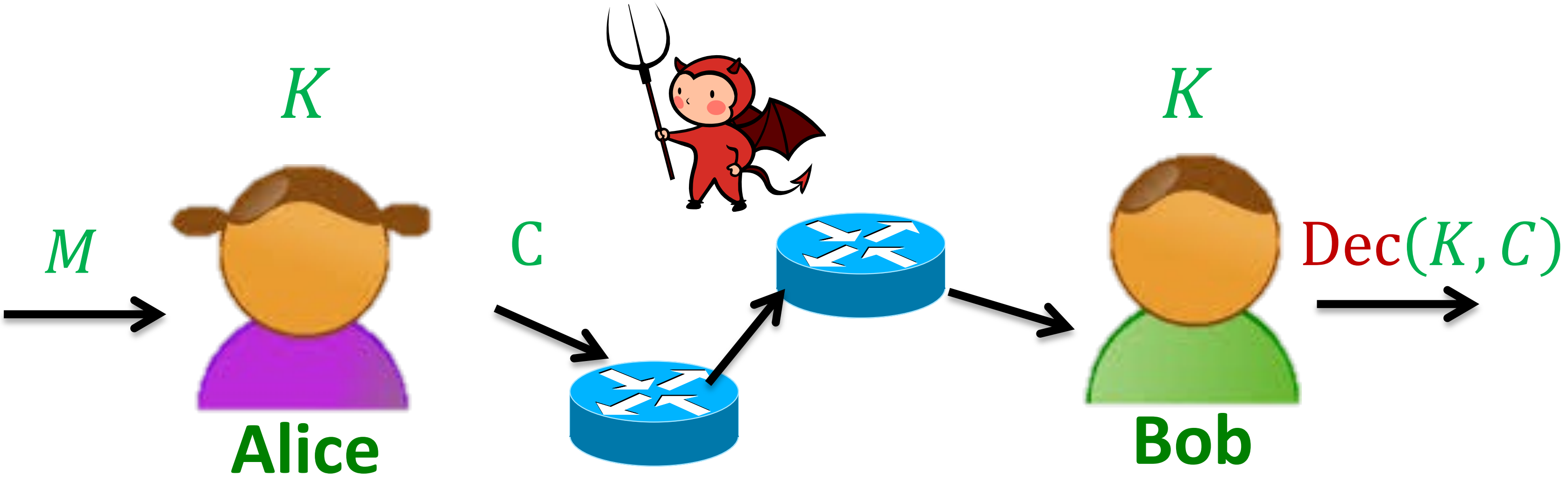
CS 177 - Computer Security

Cryptography II

Is confidentiality everything we want?



K

M

Alice

C

K

Bob

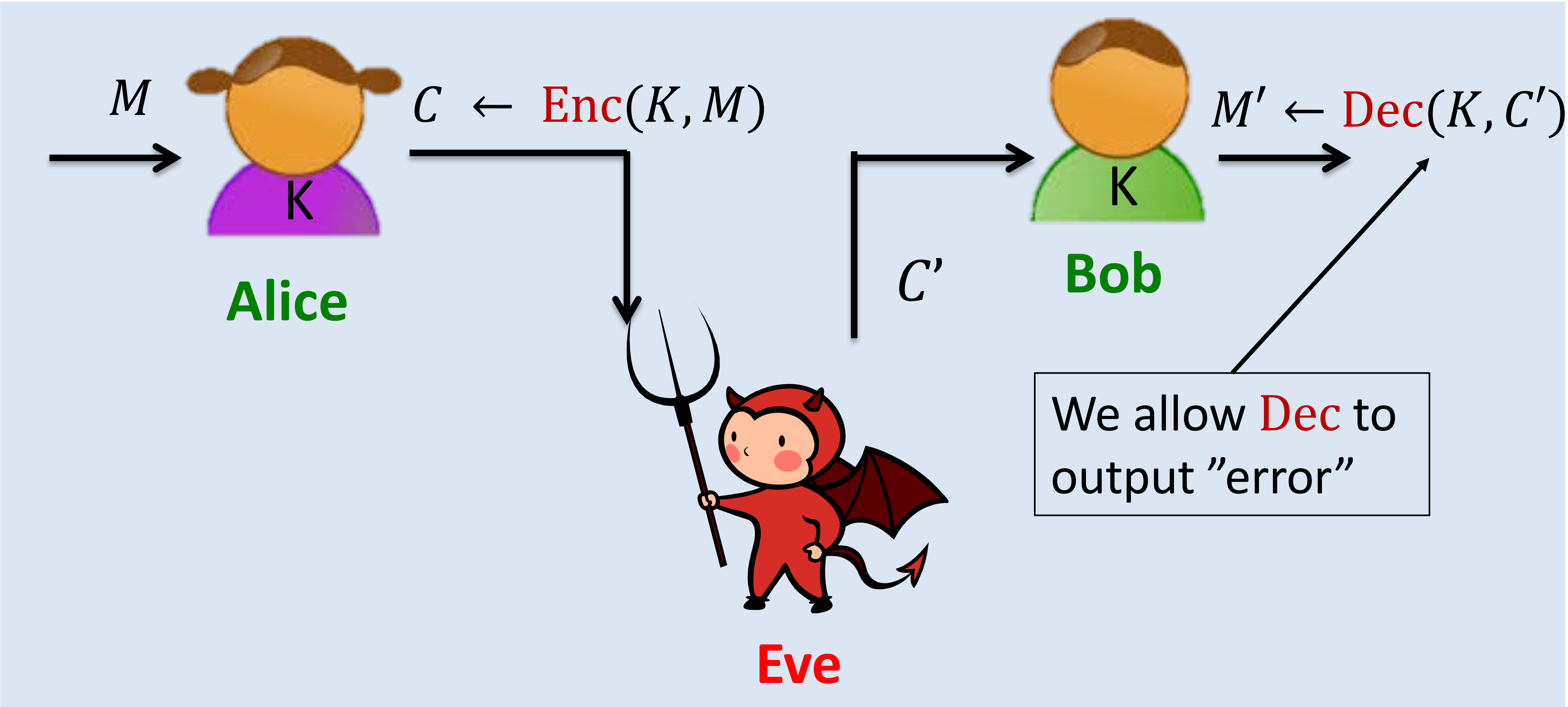
$\mathrm{Dec}(K, C)$

Confidentiality is not the only goal

We also want to make sure that the encryption scheme guarantees **integrity**

Imagine Eve tampers with ciphertext sent by Alice to Bob, then Bob must be able to detect it!

Encryption Integrity – Abstract scenario



M

Alice

K

$C \leftarrow \mathrm{Enc}(K, M)$

Eve

C'

Bob

K

$M' \leftarrow \mathrm{Dec}(K, C')$

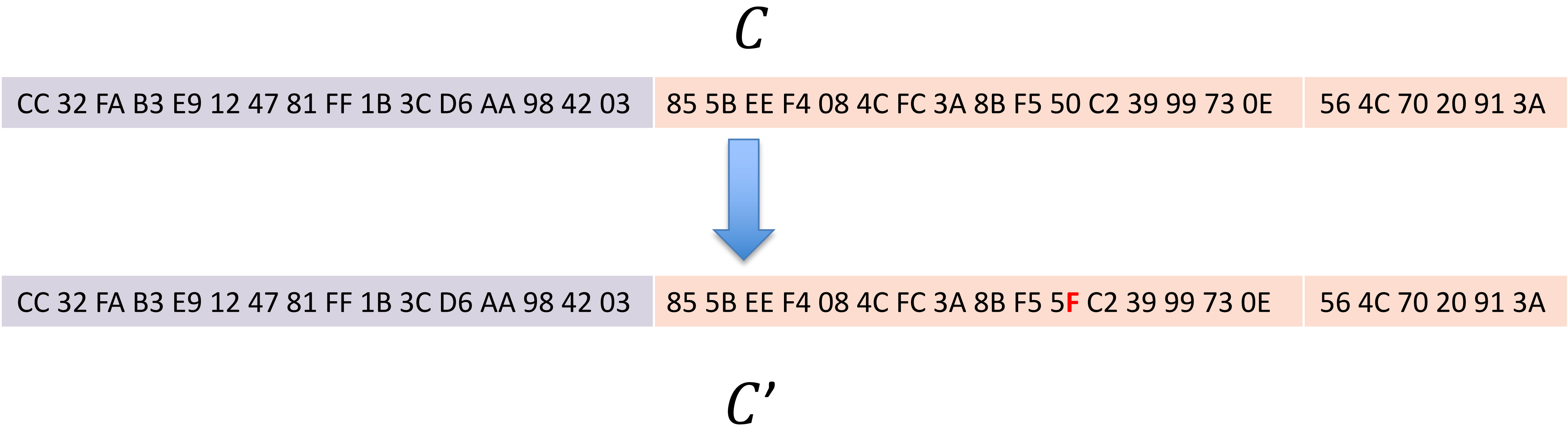
We allow Dec to output ”error”

Scheme satisfies **integrity** if it is unfeasible for Eve to send C' not previously sent by Alice such that $\mathrm{Dec}(K, C') \neq \mathrm{error}$

CTR and Integrity

Back to CTR example, imagine Eve sees the following ciphertext

[remember: it encrypts “Hello CS177 students!”, but Eve does not know this]



C

CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 03 | 85 5B EE F4 08 4C FC 3A 8B F5 50 C2 39 99 73 0E | 56 4C 70 20 91 3A

CC 32 FA B3 E9 12 47 81 FF 1B 3C D6 AA 98 42 03 | 85 5B EE F4 08 4C FC 3A 8B F5 5**F** C2 39 99 73 0E | 56 4C 70 20 91 3A

C'

Eve just changed four bits from 0 to 1, and sends C' to Bob. Bob attempts to decrypt. What does he get?

CTR and Integrity – cont’d

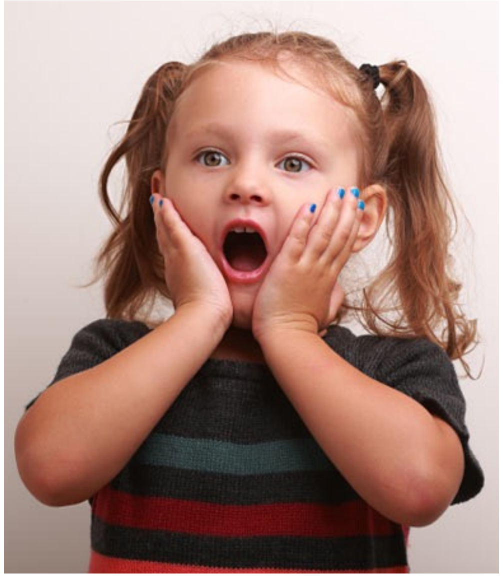
85 5B EE F4 08 4C FC 3A 8B F5 5**F** C2 39 99 73 0E	56 4C 70 20 91 3A

⊕

CD 3E 82 98 67 6C BF 69 BA C2 67 E2 4A ED 06 6A	33 22 04 53 B0 3A

Bob decrypts by adding the mask back

48 65 6C 6C 6F 20 43 53 31 37 3**8** 20 73 74 75 64	65 6E 74 73 21 00



Which is the ASCII encoding for “Hello CS17**8** students!”

What happened? Eve flipped a few bits and produced a valid encryption for something that Alice never meant to send. **NO integrity!**

Important message

“Classical” modes of operation like CTR and CBC never guarantee integrity, and should never be used by themselves.

Authenticated Encryption

AE = confidentiality + integrity

One of the trickiest topics in cryptography

- Many mistakes here have led to attacks
- Badly treated by current textbooks
- Misunderstanding is historically rooted

Central tool to achieve integrity: **Message-authentication codes** (MACs)

Message Authentication

Message Authentication Code (MAC) is an efficient algorithm that takes a secret key, a string of arbitrary length, and outputs an (unpredictable) short output/digest.

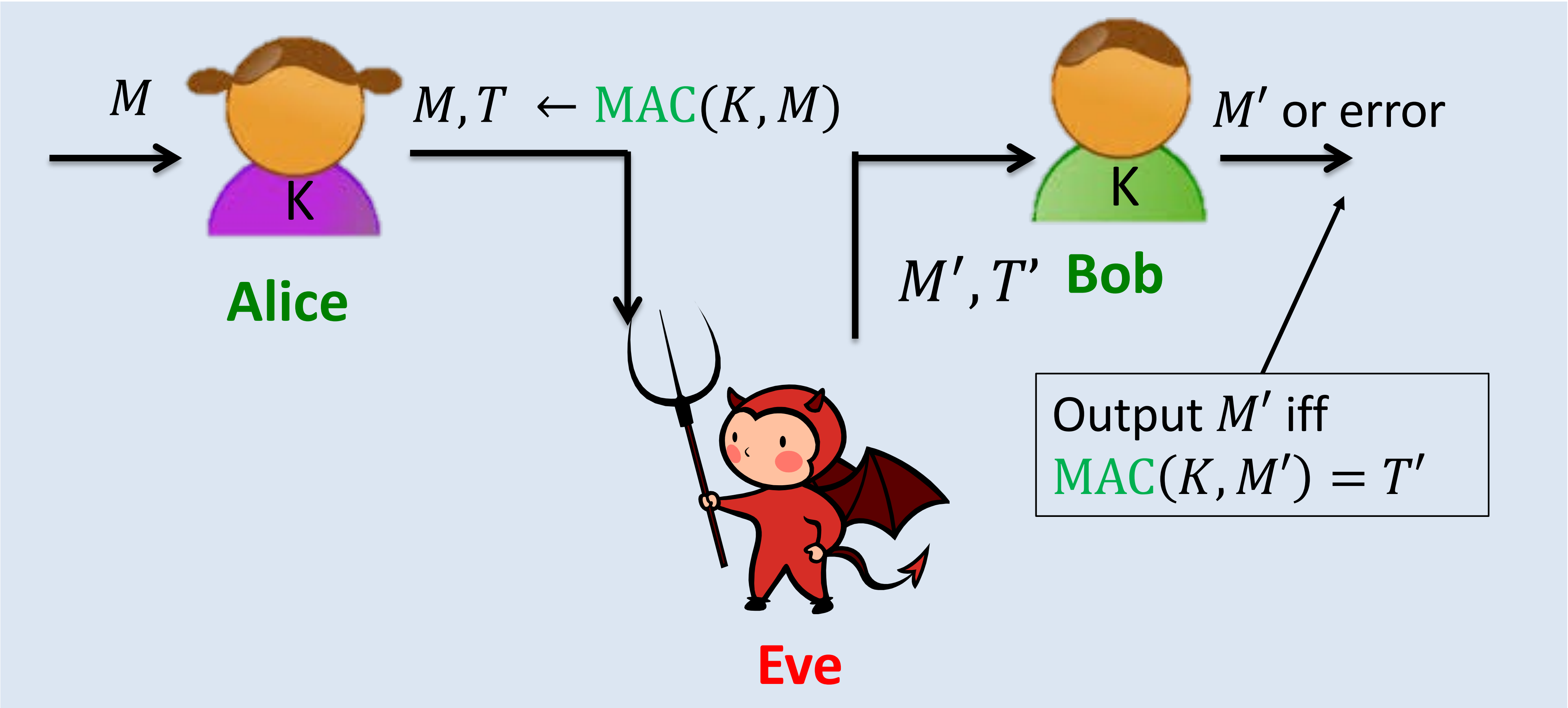
$$\mathrm{MAC}: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$$

$$\mathrm{MAC}(K, M) = \mathrm{MAC}_K(M) = T$$

message **tag**

key

Message Authentication – Scenario



M

$M, T \leftarrow \mathrm{MAC}(K, M)$

K

Alice

M', T’

Bob

K

M' or error

Output M' iff $\mathrm{MAC}(K, M') = T'$

Eve

MAC satisfies **unforgeability** if it is unfeasible for Eve to let Bob output M' not previously sent by Alice.

MAC example

Note: No encryption in this example, this is only about integrity!

M = “Hello CS177 students!”

$T = \mathrm{MAC}(K, M) =$ 5f 68 18 21 b7 f5 4f b1 10 3d fd fa 89 0e ca 1d 42 10 7d 2f

M' = “Hello CS17**8** students!”

$T' = \mathrm{MAC}(K, M') = ???$

Any guess likely incorrect!

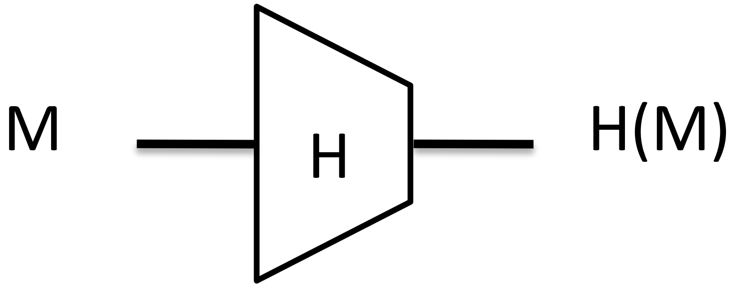
Baseline

- Knowing the key allows to compute/recompute the message tag.
- Not knowing the key makes the tag unpredictable (unless we have seen it already).

Hash functions and message authentication

Many MACs are built from **cryptographic hash functions**

Hash function H maps arbitrary bit string to fixed length string of size m



M

H

H(M)

MD5:	m = 128 bits
SHA-1:	m = 160 bits
SHA-256:	m = 256 bits
SHA-3:	m >= 224 bits

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)

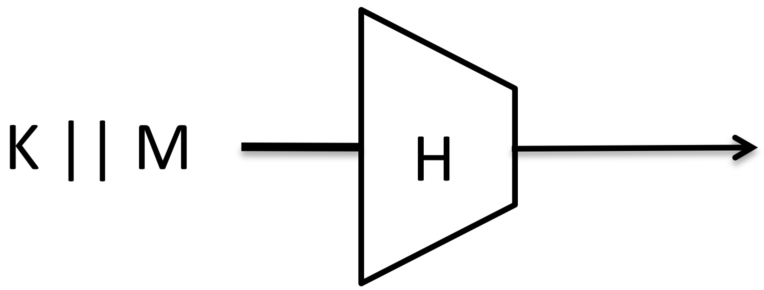
Hash-function side-note

- MD5 and SHA-1 are broken
  - Never use them in anything you are going to develop and/or deploy!
  - https://www.youtube.com/watch?v=NbHL0SYlrSQ
- SHA-256, SHA-512, SHA-3, BLAKE2 all ok
- SHA-256/SHA-512 most widely used

Message authentication with hash functions

Goal: Use a hash function H to build MAC

MAC(K, M) = H(K || M)



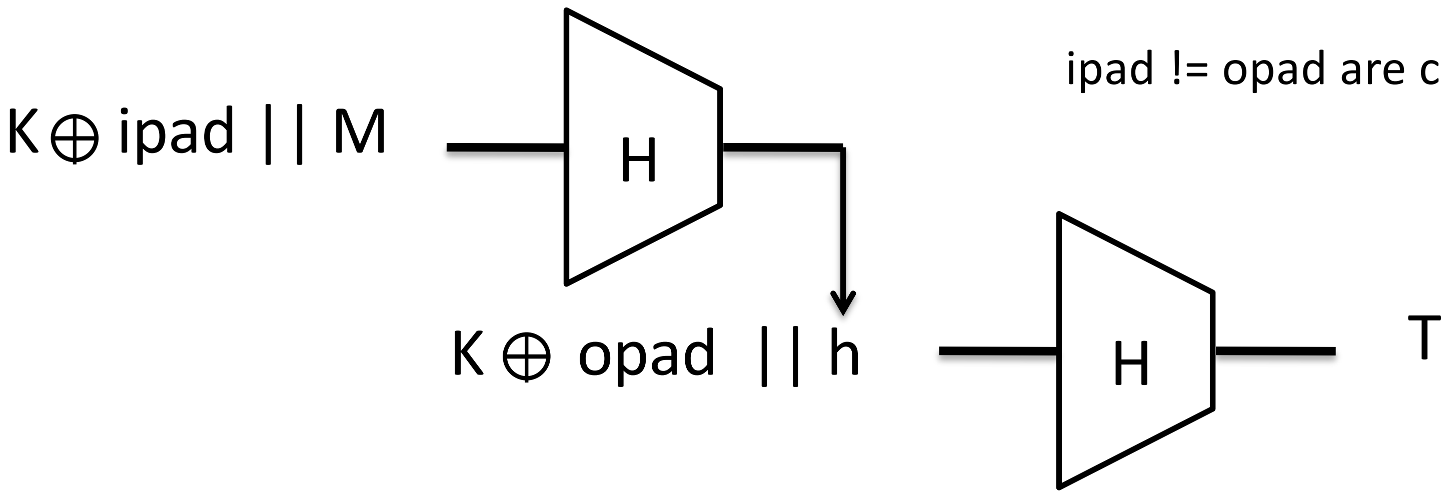
In other words: The MAC is the hash of the concatenation of the key and the message.

- Good option for SHA-3 / BLAKE2
- Completely insecure for SHA-256/SHA-512
- Length extension attack
  - from hash(m_1), it is easy to compute hash($m_1 \| m_2$)

Message authentication with hash functions

Goal: Use a hash function H to build MAC

HMAC(K,M) defined by:



ipad != opad are constants

Unforgeability holds if H is secure in some well-defined sense

No attacks in particular for SHA-256/SHA-512

Important

Hash function $\neq$ MAC

A hash function takes no key, a MAC is a secret-key primitive

Helpful intuition: A MAC is like a hash function which can only be evaluated by those having the secret key.

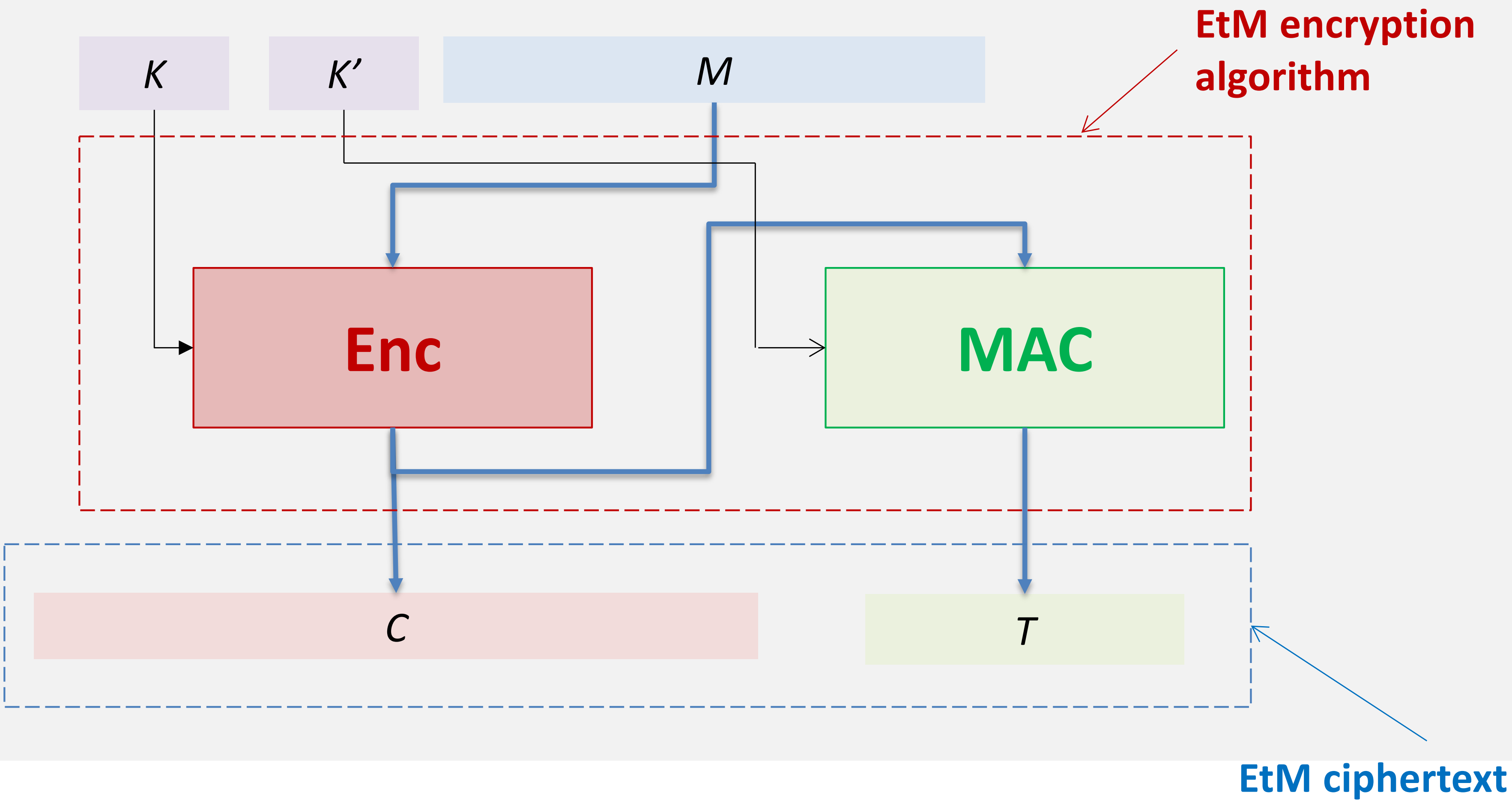
How to achieve integrity?

Combine a MAC and a semantically secure encryption scheme!

Best solution: **Encrypt-then-MAC**

Encrypt-then-MAC

EtM key consists of two keys
(one for Enc, one for MAC)



Decryption: Given $C^* = (C, T)$, first check T valid tag for C using K'

- If so, decrypt C, and output result
- If not, output "error"

Encrypt-then-MAC – why is it secure?

EtM is secure as long as encryption scheme is semantically secure, and MAC is unforgeable!

Integrity. If the attacker sees $C^* = (C, T)$, and wants to change this to a valid $C^{**} = (C', T')$ where $C' \neq C$, then it needs to forge the MAC, i.e., produce a new tag T' for C'.

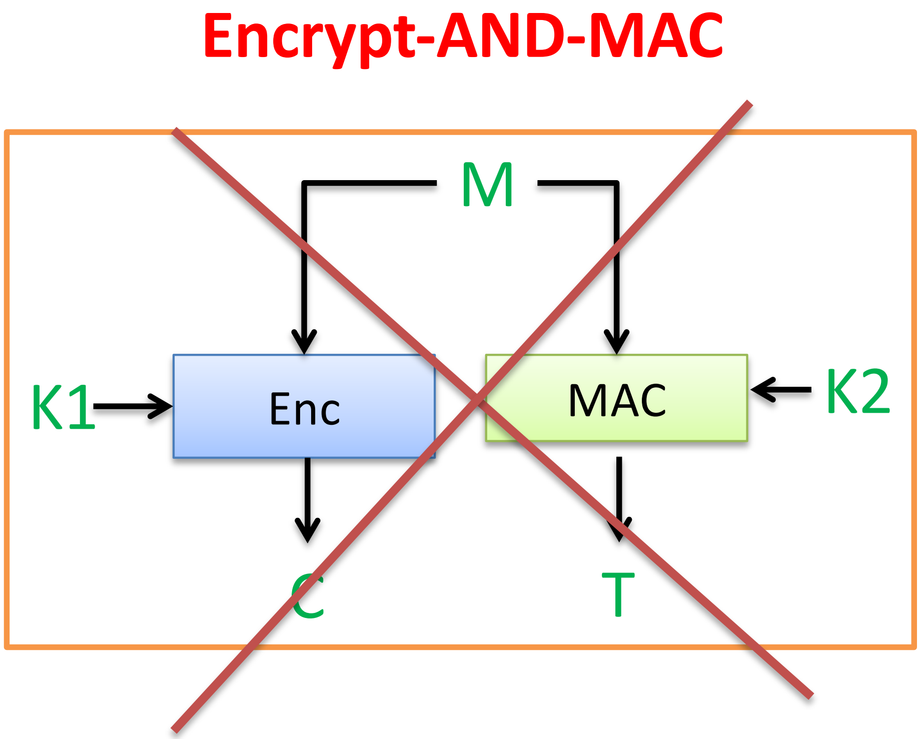
Confidentiality. $C^* = (C, T)$ does not leak more information about plaintext than C, because T is computed from C directly, and does not add extra information about plaintext.

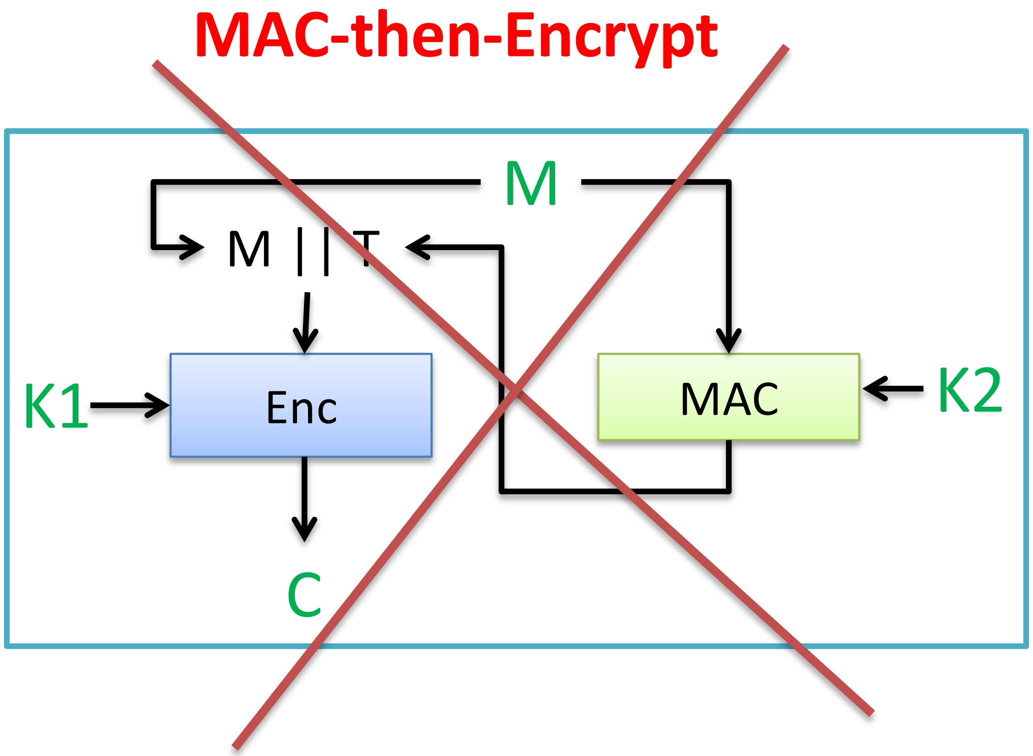
Encrypt-then-MAC

Valid combinations are e.g.

{AES-CTR, AES-CBC} + {SHA-256-HMAC, SHA-512-HMAC}

Authenticated Encryption – Bad Solutions





Still, they are used all over the place, but just don’t use them

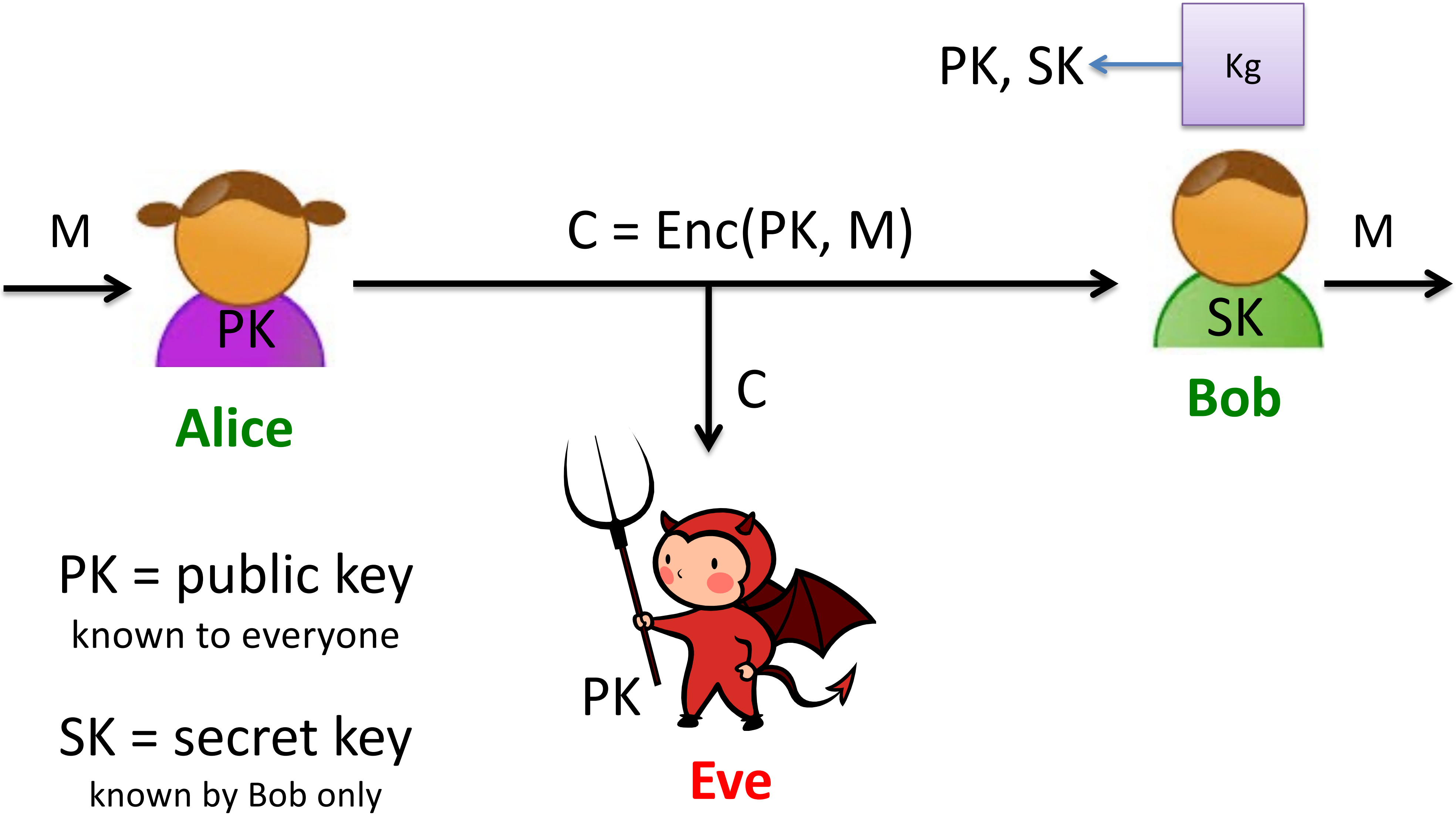
Public-key Encryption Scheme

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* $\mathrm{C} \leftarrow \mathrm{Enc}(PK, M)$
- **Decryption algorithm** Dec, is such that

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

Asymmetric Encryption
(aka public-key encryption (PKE))



PK, SK ← Kg

M → Alice (PK) → C = Enc(PK, M) → Bob (SK) → M

C → Eve (PK)

PK = public key

known to everyone

SK = secret key

known by Bob only

The RSA Algorithm

- Rivest, Shamir, Adleman 1978
- Garnered them a Turing award

RSA math

typically referred to as "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 = 13, gives	N = 91
p = 17, q = 53, gives	N = 901

Modular arithmetic – Basic sets

$$\boldsymbol{Z}_N \;=\; \{0,1,2,3,\ldots,N-1\}$$

$$\boldsymbol{Z}_N^* \;=\; \{\, i \mid \gcd(i,N) \;=\; 1 \,\}$$

$\gcd(X,Y) \;=\; 1$ if greatest common divisor of X,Y is 1

Basic sets – Example

$$\boldsymbol{Z}_N^* = \{ i \mid \gcd(i, N) = 1 \}$$

$N = 13$ $\boldsymbol{Z}_{13}^*$ = { 1,2,3,4,5,6,7,8,9,10,11,12 }

$N = 15$ $\boldsymbol{Z}_{15}^* = \{ 1,2,4,7,8,11,13,14 \}$

Def. $\varphi(N) = |\boldsymbol{Z}_N^*|$ (Euler's totient function)

$\boldsymbol{Z}_{\varphi(15)}^* = \boldsymbol{Z}_8^* = \{ 1,3,5,7 \}$

$\varphi(13) = 12$

$\varphi(15) = 8$

Fact. If p, q are distinct primes, $\varphi(p \times q) = (p-1)\times(q-1)$

Modular Arithmetic

Fact. For any a, N with $N > 0$, there exists unique q,r such that

$$a = Nq + r \qquad \text{and} \qquad 0 \leq r < N$$

Def. $a \bmod N = r \in \boldsymbol{Z}_N$

17 mod 15 = 2

105 mod 15 = 0

RSA Math

Lemma. Suppose $e, d \in \boldsymbol{Z}^*_{\varphi(N)}$ satisfy $ed = 1 \left(\mathrm{mod}\ \varphi(N)\right)$, then for any $x \in \boldsymbol{Z}_N$ we have that

$$(x^e)^d = x^{ed} = x \ \text{[mod n]}$$

Euler's Theorem: $a^{\varphi(n)} \equiv 1 \pmod{n}$

N = 15, e = 3, d = 3 [ed mod $\varphi(N)$ = ed mod 8 = 1]

x	1	2	4	7	8	11	13	14
y = x^3 mod 15	1	8	4	13	2	11	7	14
y^3 mod 15	1	2	4	7	8	11	13	14

RSA Encryption

$PK = (N, e) \quad SK = (N, d)$ with $ed = 1 \pmod{\varphi(N)}$

$$\mathrm{Enc}((N, e), M) = M^e \bmod N$$
$$\mathrm{Dec}((N, d), C) = C^d \bmod N$$

Messages / ciphertexts are elements of $\boldsymbol{Z}_N$

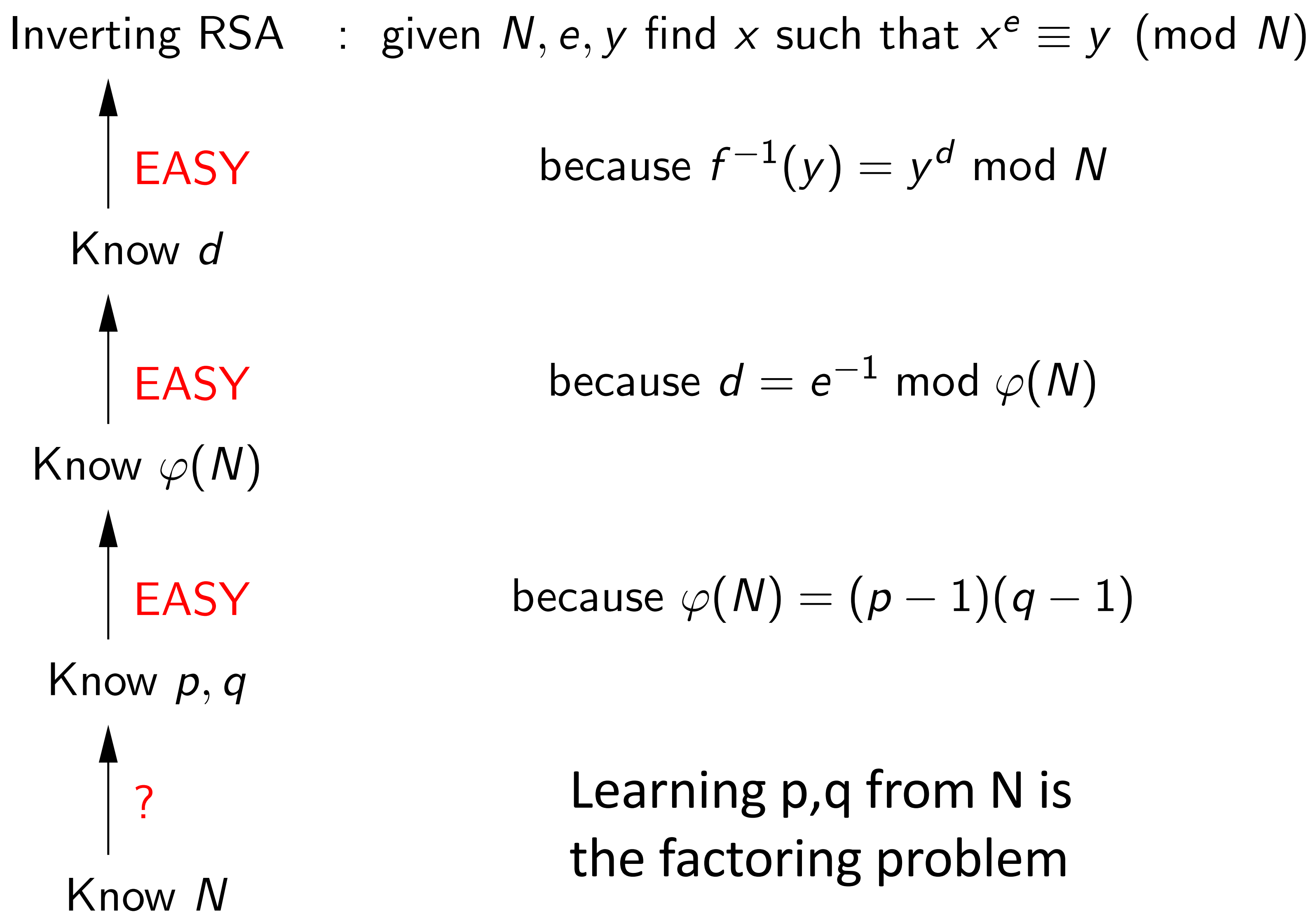
But how do we find suitable N, e, d?

Given $\varphi(N) = (p-1)(q-1)$, choose e first, and then choose d such that $ed = 1 \pmod{\varphi(N)}$ (An efficient algorithm for this exists.)

Security of “plain” RSA

- Passive adversary sees N, e, and C
- Attacker would like to invert C (get M, or d)
- Possible attacks?

$$\begin{tikzcd}
M \arrow[rrr, bend left, "\text{easy given N,e}"] & & & \mathrm{C{=}Enc}((N,e),M) \arrow[lll, bend left, "\begin{array}{c}\text{hard given N,e}\\ \text{easy given N,d}\end{array}"]
\end{tikzcd}$$



Inverting RSA : given N, e, y find x such that $x^e \equiv y \pmod{N}$

EASY

because $f^{-1}(y) = y^d \bmod N$

Know d

EASY

because $d = e^{-1} \bmod \varphi(N)$

Know $\varphi(N)$

EASY

because $\varphi(N) = (p-1)(q-1)$

Know p, q

?

Learning p,q from N is the factoring problem

Know N

We don’t know if inverse is true, whether inverting RSA implies ability to factor, but they are equivalent in practical terms

Factoring composites – How hard?

- What is p,q for N = 901?

```
Factor(N):
for i = 2 , … , sqrt(N) do
    if N mod i = 0 then
        p = i
        q = N / p
        Return (p,q)
```

If you do this, as soon as you reach 17, you will learn that 901 = 17 x 53

Woops… we can always factor

But not always efficiently:
Run time is sqrt(N)

$\mathcal{O}(\text{sqrt}(N)) = \mathcal{O}(e^{0.5 \ln(N)})$

Factoring records

Algorithm	Year	Algorithm	Time
RSA-400	1993	QS	830 MIPS years
RSA-478	1994	QS	5000 MIPS years
RSA-515	1999	NFS	8000 MIPS years
RSA-768	2009	NFS	~2.5 years

RSA-x is an RSA challenge modulus of size x bits

Hybrid Encryption

Normally, public-key encryption is orders of magnitude slower than secret-key encryption.

- E.g., AES-NI instructions give CPU-level support for AES encryption/decryption

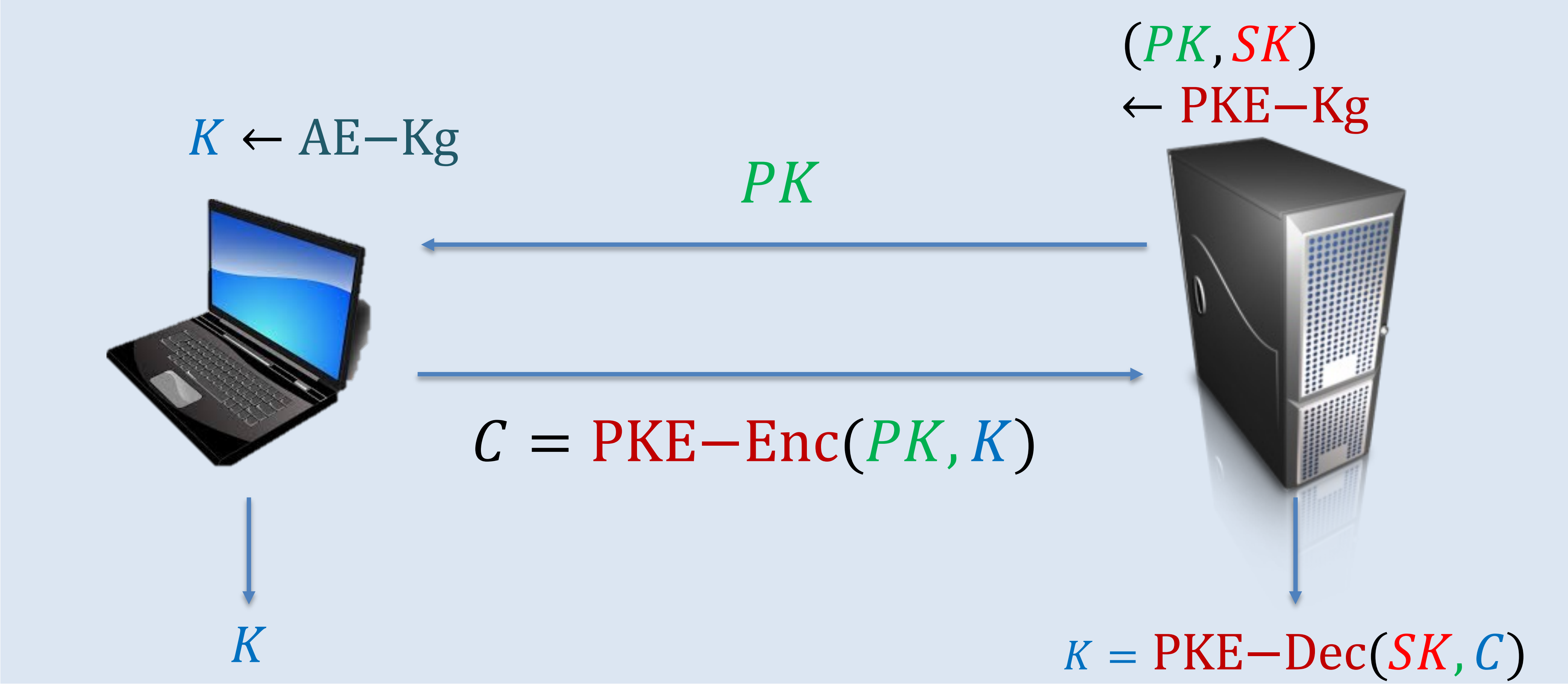
How do we deal with this?

Solution: Use public-key encryption only to agree on a secret key. Then, use secret-key encryption

Key Exchange

PKE scheme $\mathrm{PKE} = (\mathrm{PKE{-}Kg}, \mathrm{PKE{-}Enc}, \mathrm{PKE{-}Dec})$,
Symmetric auth. encryption scheme $\mathrm{AE} = (\mathrm{AE{-}Kg}, \mathrm{AE{-}Enc}, \mathrm{AE{-}Dec})$

Goal: Client and server agree on key K for AE



$(PK, SK) \leftarrow \mathrm{PKE{-}Kg}$

$K \leftarrow \mathrm{AE{-}Kg}$

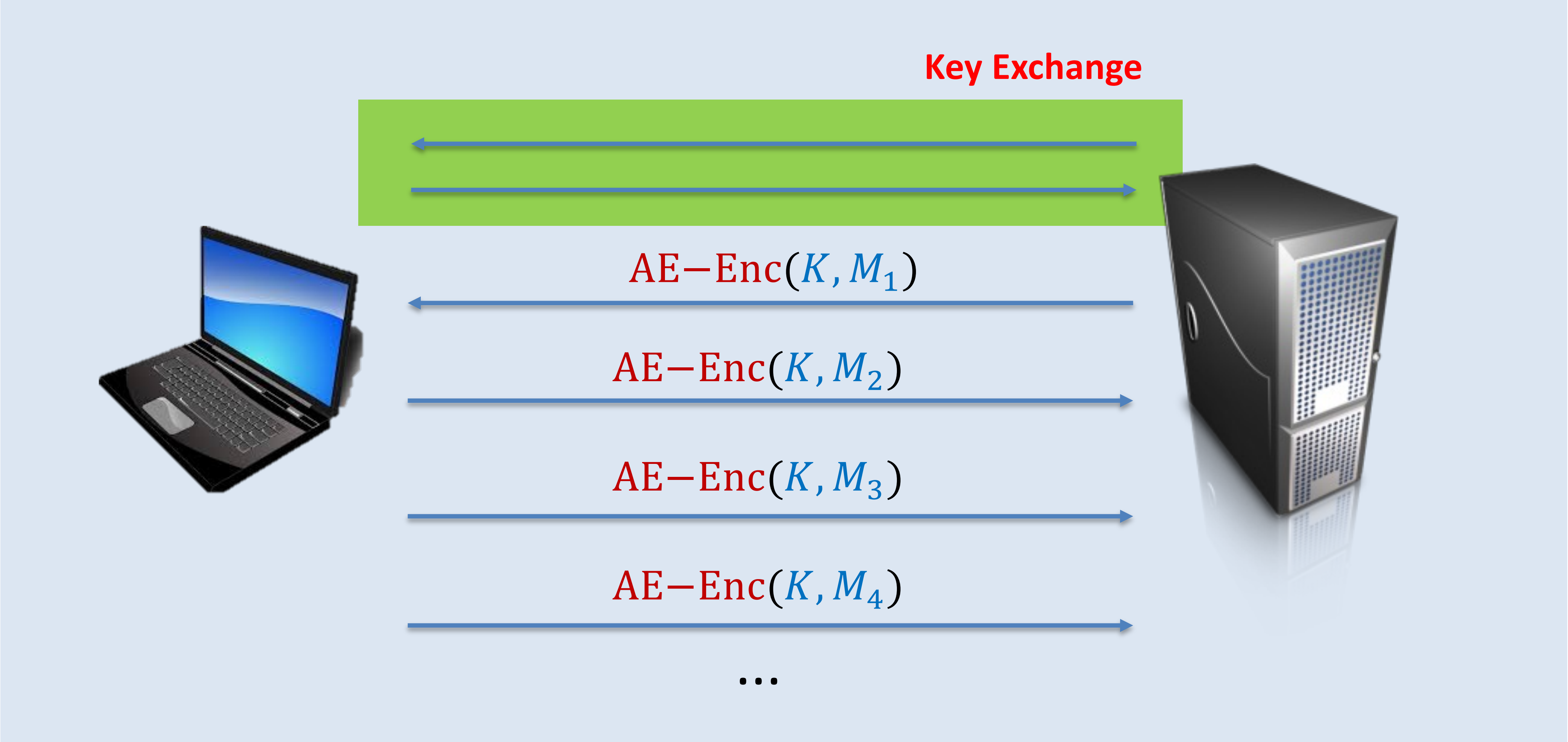
PK

$C = \mathrm{PKE{-}Enc}(PK, K)$

K

$K = \mathrm{PKE{-}Dec}(SK, C)$

Hybrid Encryption



Key Exchange

$\mathrm{AE{-}Enc}(K, M_1)$

$\mathrm{AE{-}Enc}(K, M_2)$

$\mathrm{AE{-}Enc}(K, M_3)$

$\mathrm{AE{-}Enc}(K, M_4)$

…

After agreeing on secret key K, the client and the server can exchange messages (very fast) using authenticated encryption

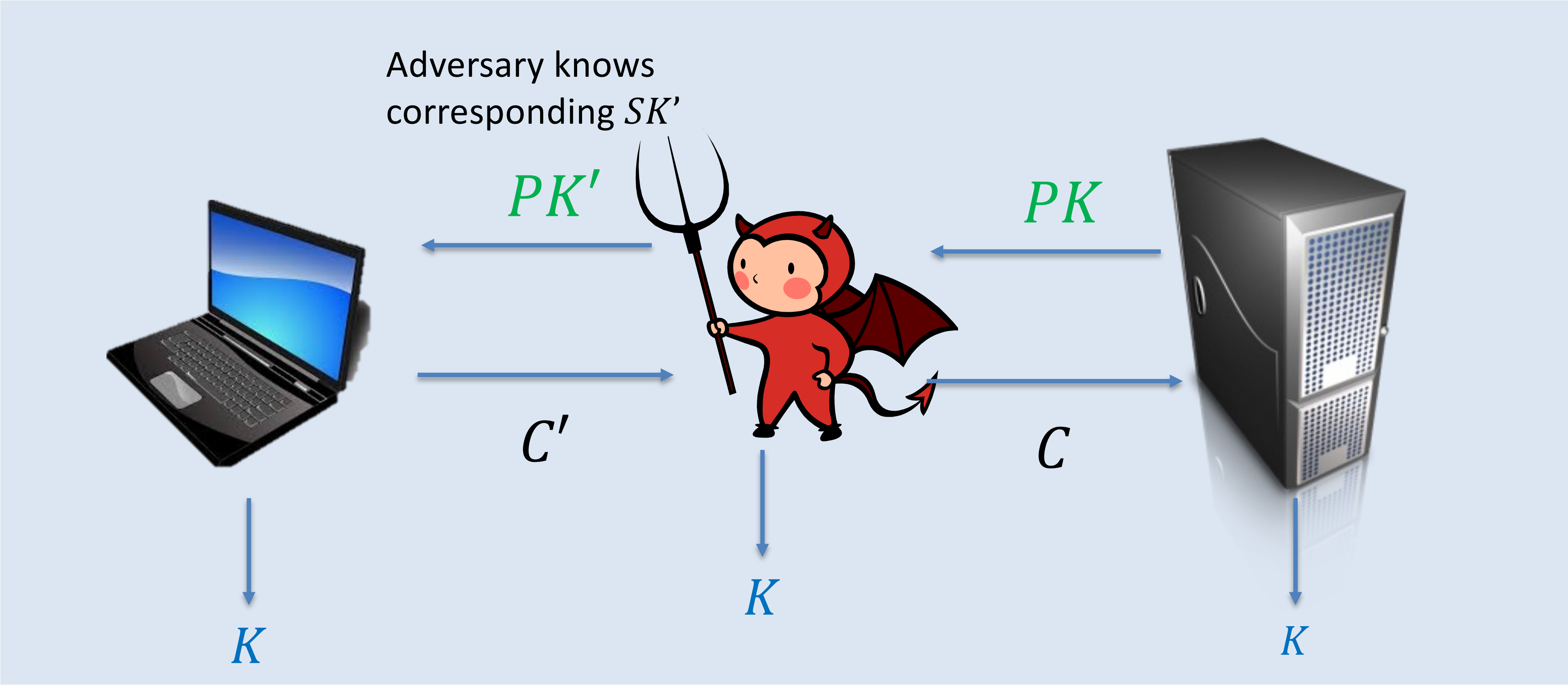
Overall structure behind TLS, SSH, etc.

Other Key Agreement Approaches

- Diffie-Hellman Key-Exchange
- Underlying mathematics based on the “discrete logarithm on elliptic curves”
- Better security, smaller bandwidth (256 bits per round vs 4096 bits for RSA)
- Main disadvantage: Less support (for now), but we are getting there – TLS 1.3

Caveat: Man-in-the-middle attacks

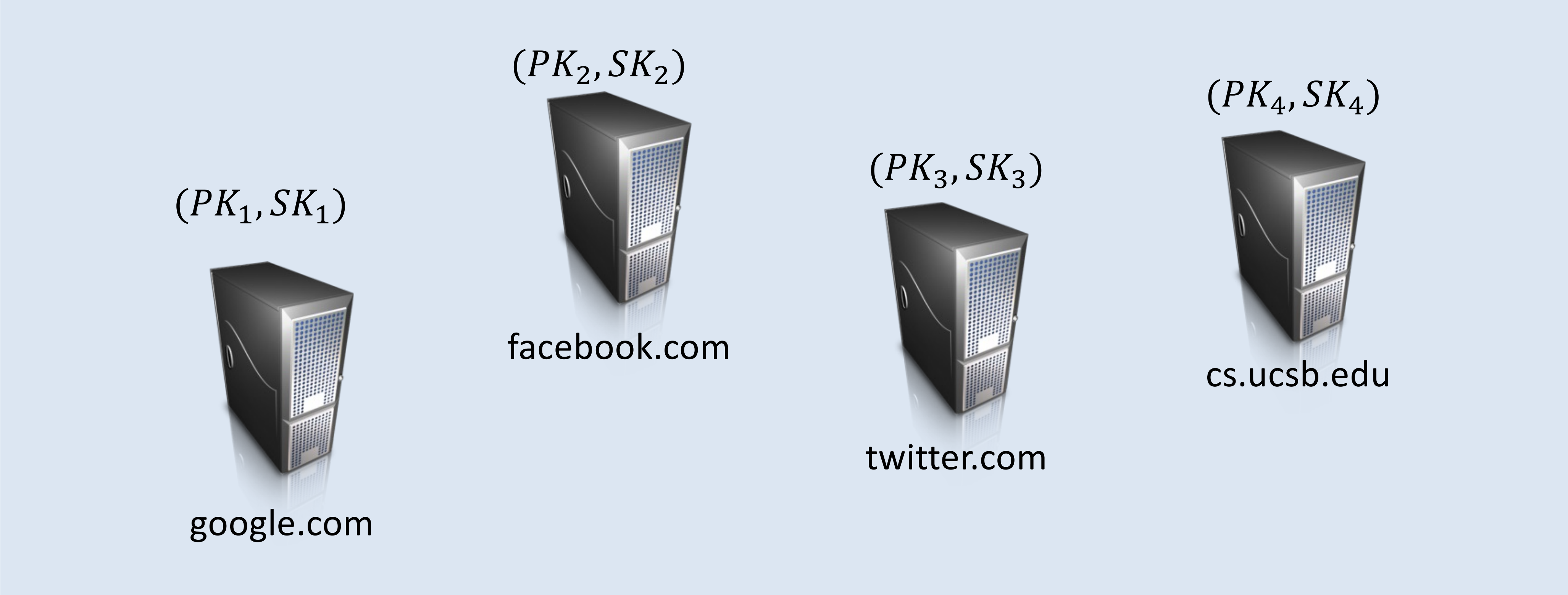
Adversary can transparently sit between client and server



Adversary now knows secret key K generated by client, as it is encrypted with her key (and she can then forward it to server, encrypting it with the server's PK)

Public-key infrastructures

Public-key cryptography enables individuals to generate their own key pairs, but how does one decide whether a (public) key is legitimate?



(PK_2, SK_2)

(PK_4, SK_4)

(PK_3, SK_3)

(PK_1, SK_1)

facebook.com

cs.ucsb.edu

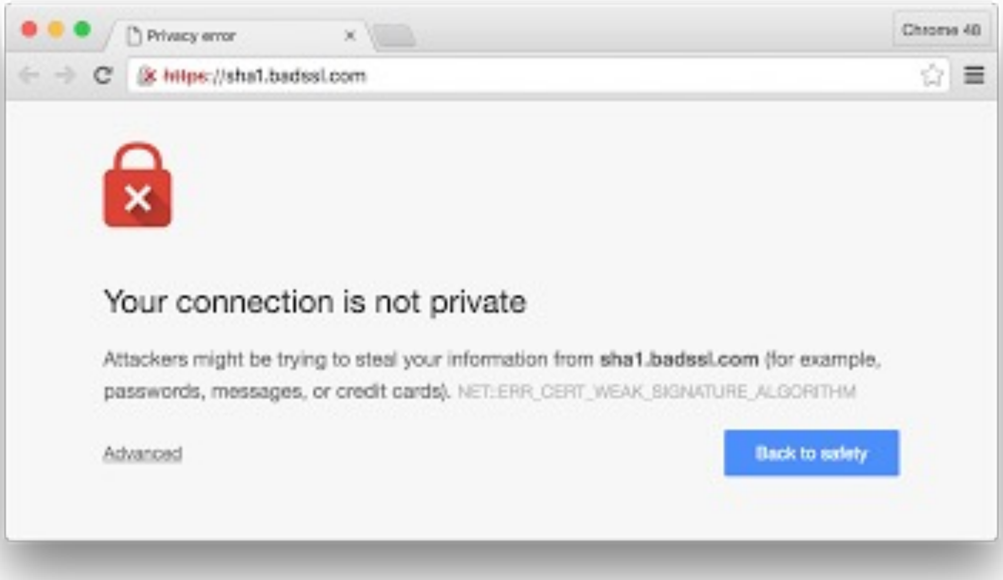
twitter.com

google.com

Example: We connect to google.com in TLS, receive PK -- how do we know whether $PK = PK_1$? (And not something else sent by a man-in-middle?)

How do we resolve this?

Modern browser’s indeed complain when public-key not trusted



Your connection is not private

Attackers might be trying to steal your information from **sha1.badssl.com** (for example, passwords, messages, or credit cards). NET::ERR_CERT_WEAK_SIGNATURE_ALGORITHM

Naïve solution

Every browser stores a list of public keys of all possible services!

keys.txt:

google.com: PK_1

facebook.com: PK_2

twitter.com: PK_3

cs.ucsb.edu: PK_4

…

Good idea?

Obvious drawbacks:

- List is huge, needs to contain one entry for every address supporting TLS
- List needs to be updated/expanded
- Issuer of the list needs to ensure that all public keys are correct!
- User needs to trust issuer

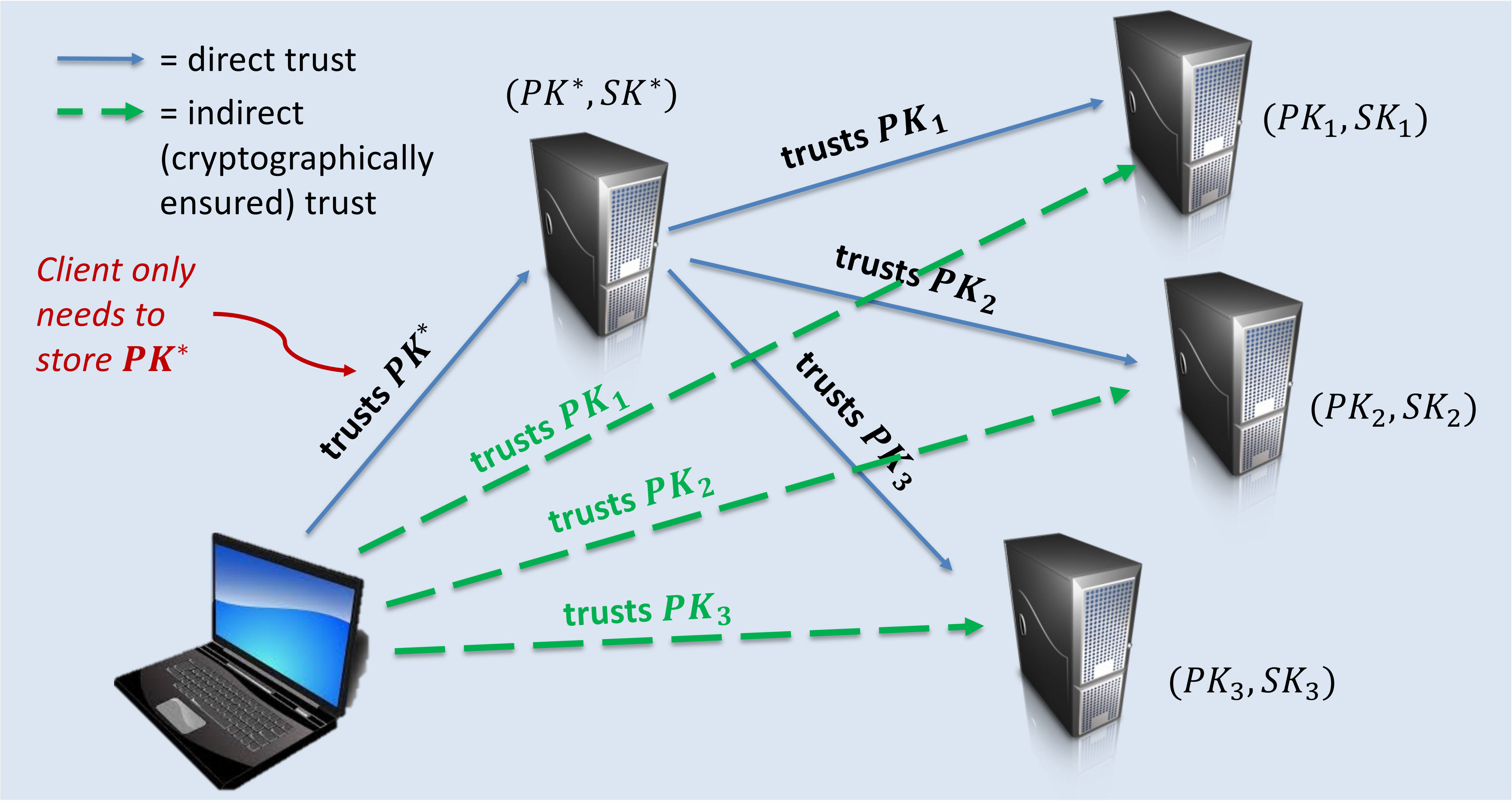
Certificates – Transferring trust

We want a mechanism that enforces the following:

> *If **A** knows that PK_B belongs to a trusted (in the eyes of **A**) entity **B**, and **B** knows that PK_C belongs to a trusted (in the eyes of **B**) entity **C**, then **A** should also trust **C** and PK_C.*

Important fact: Normally, trust can only be transferred digitally, but never created. Initially, trust needs to be established off-band.

Certificates – Transferring trust



→ = direct trust

⇢ = indirect (cryptographically ensured) trust

Client only needs to store $\boldsymbol{PK^*}$

(PK^*, SK^*)
(PK_1, SK_1)
(PK_2, SK_2)
(PK_3, SK_3)
trusts $\boldsymbol{PK^*}$
trusts $\boldsymbol{PK_1}$
trusts $\boldsymbol{PK_2}$
trusts $\boldsymbol{PK_3}$
trusts $\boldsymbol{PK_1}$
trusts $\boldsymbol{PK_2}$
trusts $\boldsymbol{PK_3}$

Digital Signatures

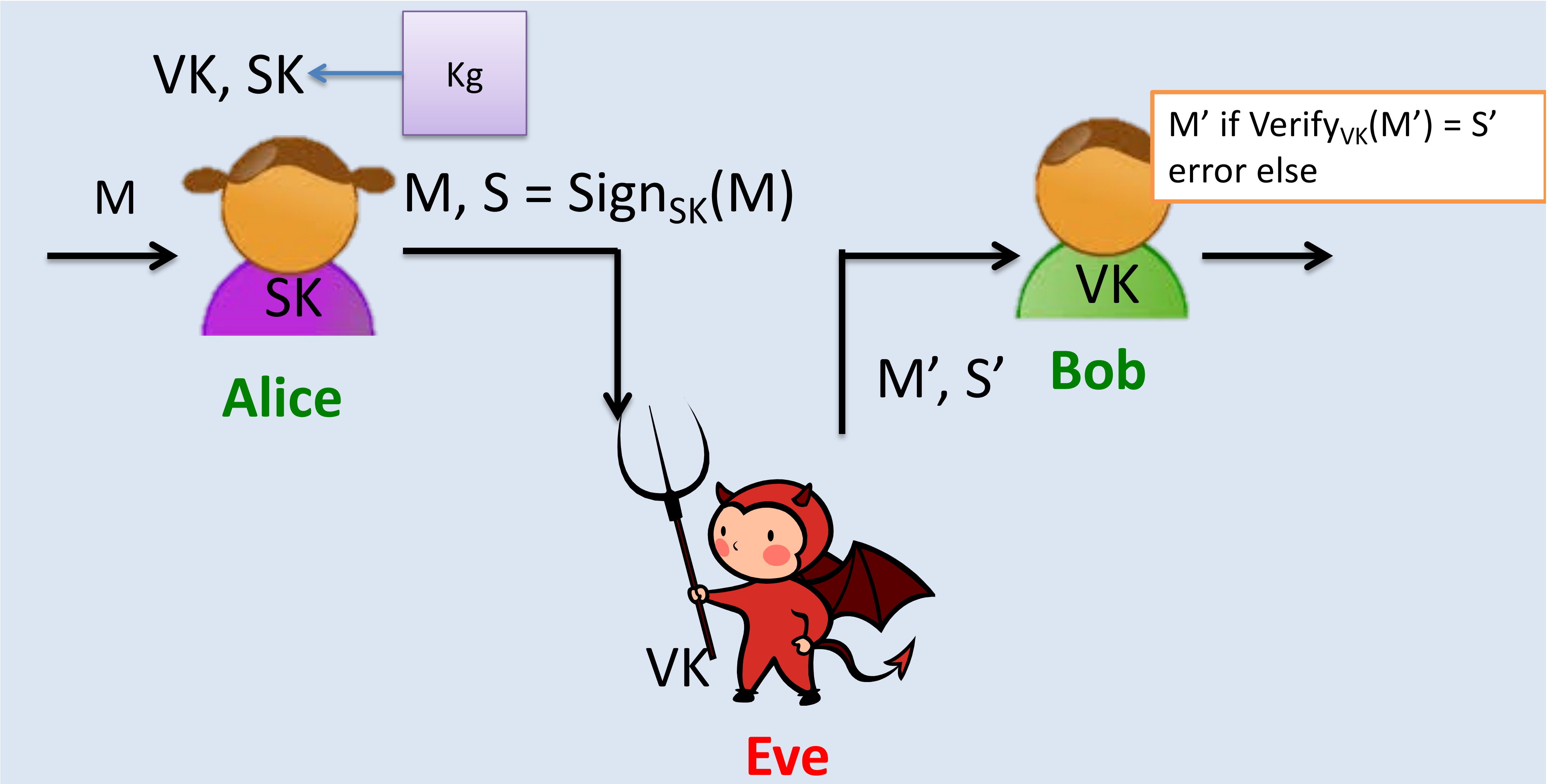
Think: The public-key version of a MAC!

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

- **Key generation algorithm** Kg, takes no input and outputs a (random) *verification key/signing key pair* (VK, SK)
- **Signing algorithm** Sign, takes input the signing key SK and the *plaintext* M, outputs *ciphertext* $S \leftarrow \mathrm{Sign}(SK, M)$
- **Verification algorithm** Verify, is such that

$$\mathrm{Verify}(VK, (M, \mathrm{Sign}(SK, M))) = \mathbf{valid}$$

Digital Signatures



VK, SK

Kg

M

SK

Alice

M, S = Sign$_{SK}$(M)

M’, S’

VK

Bob

M’ if Verify$_{VK}$(M’) = S’
error else

VK

Eve

Unforgeability: Eve must not be able to generate valid S’ for M’ not sent by Alice, even given VK

Digital Signatures Instantiations

Most commonly adopted: RSA Signatures

Hash the message, and apply RSA decryption i.e.,

- SK = (N, d)
- VK = (N, e)
- Sign(SK, M) = $H(M)^d$ mod N

Further common options: ECDSA and Ed25591

- rely on elliptic curves and give much smaller signatures and keys