Crypto
Part 3 of 3

CMSC 23200/33250, Winter 2020, Lecture 5

David Cash and Blase Ur

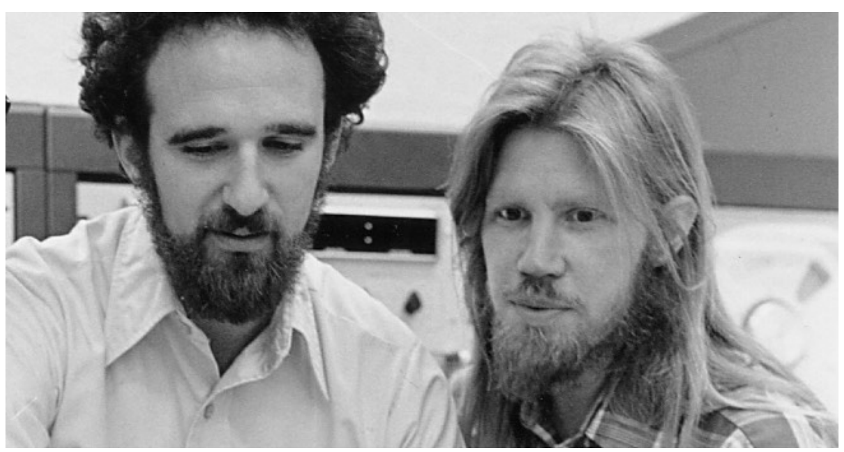
University of Chicago

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?

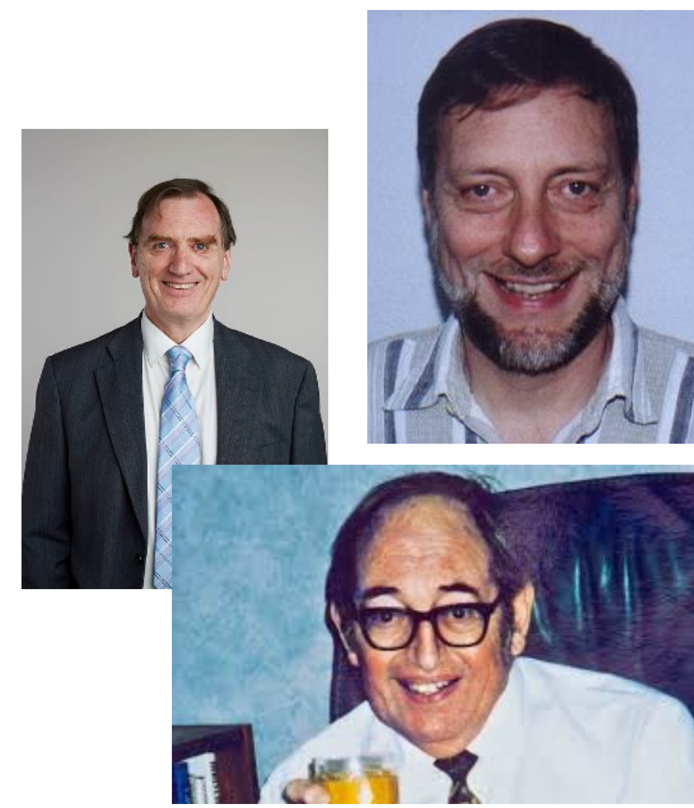


Diffie and Hellman
in 1976: **Yes!**

Turing Award, 2015,
+ Million Dollars

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

Turing Award, 2002,
+ no money

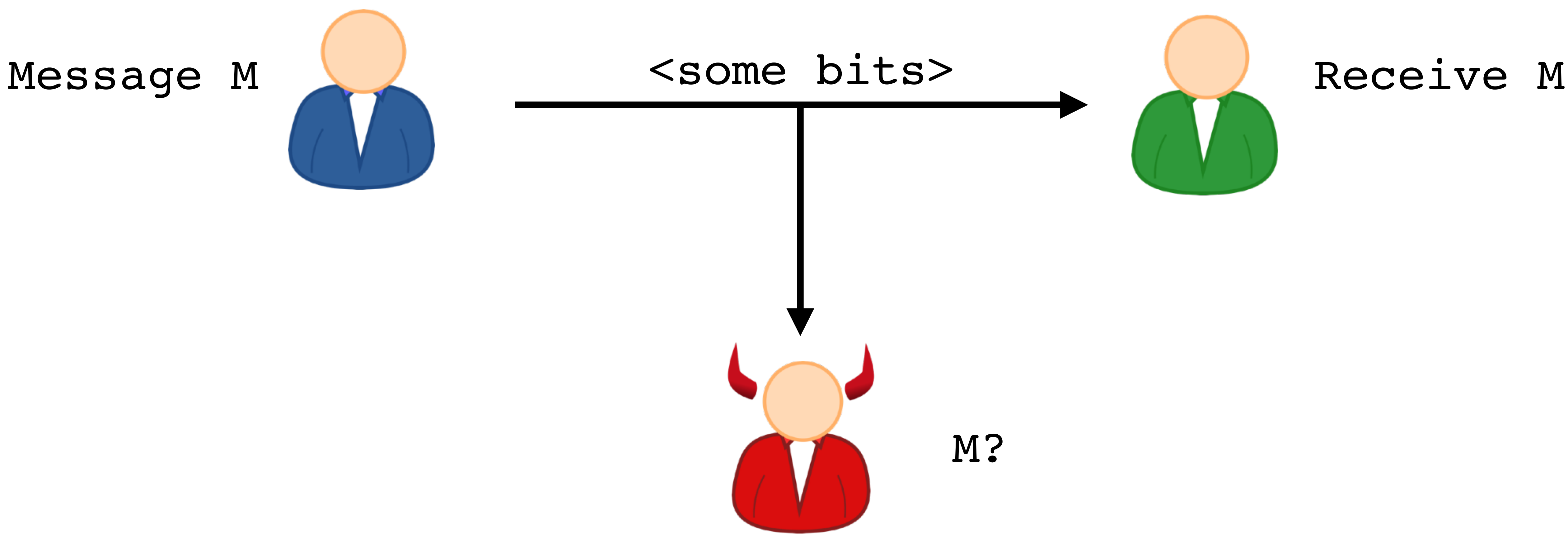


Cocks, Ellis, Williamson
in 1969, at GCHQ:
Yes, we know about both…

Pat on the back?

Public-Key Encryption

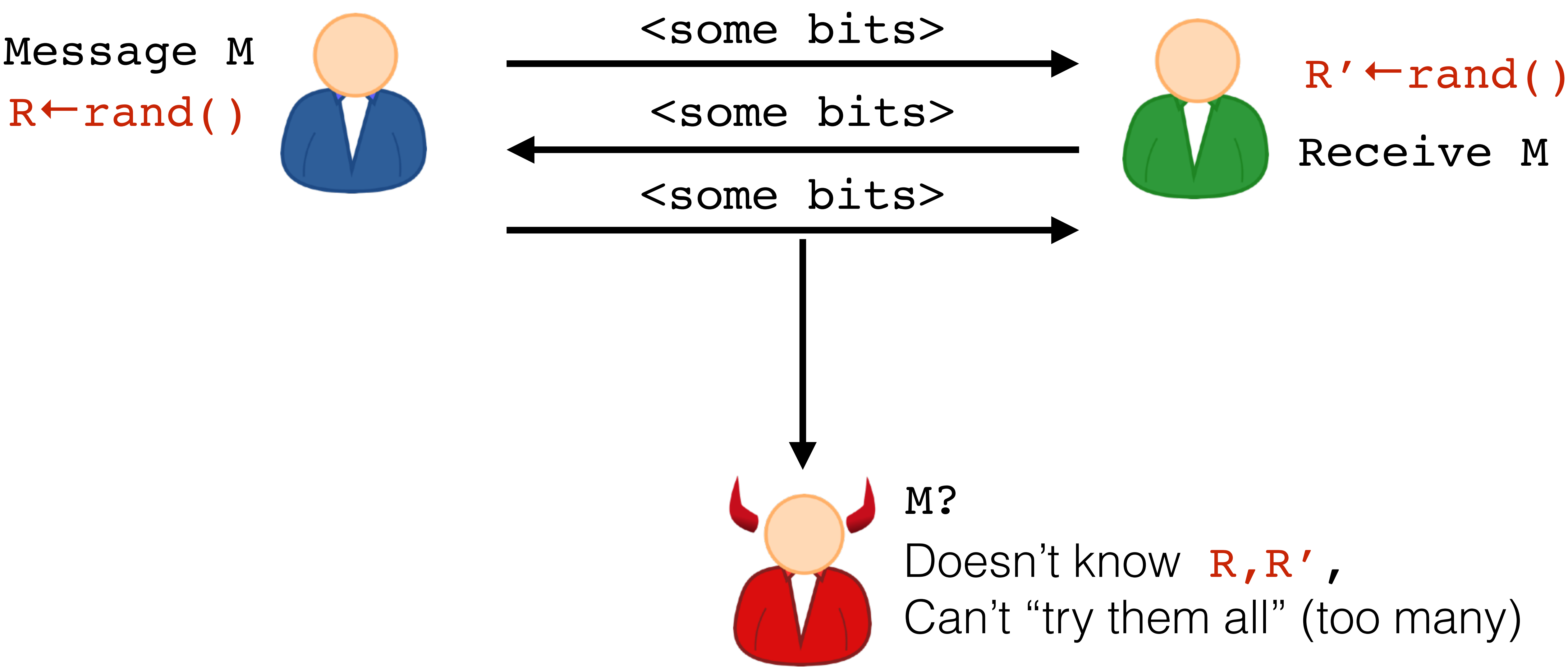
Basic question: If two people are talking in the presence of an eavesdropper, and they don't have pre-shared a key, is there any way they can send private messages?



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 <u>public-key encryption scheme</u> consists of three algorithms **Kg**, **Enc**, and **Dec**

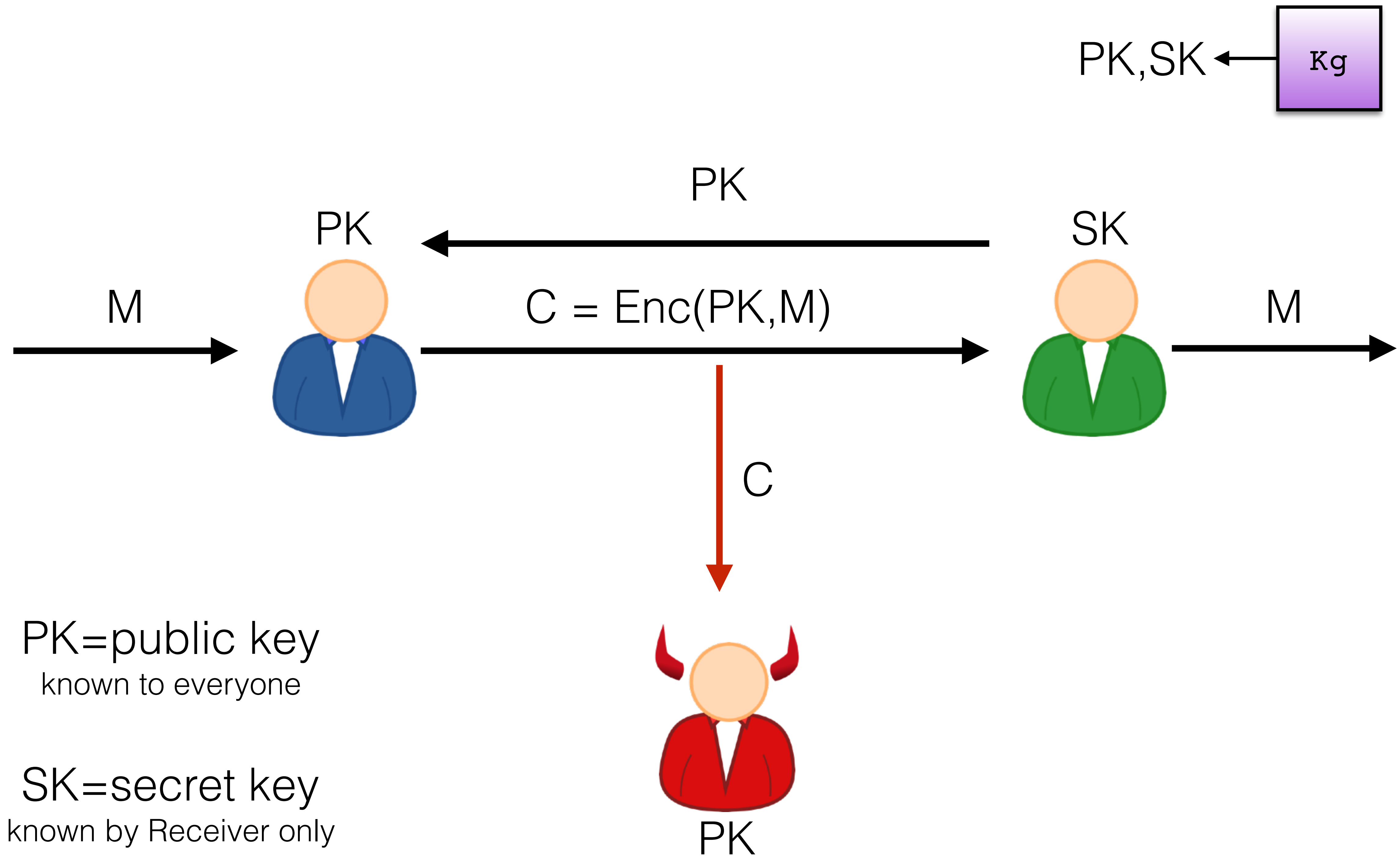
- <u>Key generation algorithm Kg</u>, takes no input and outputs a (random) public-key/secret key pair **`(PK,SK)`**

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

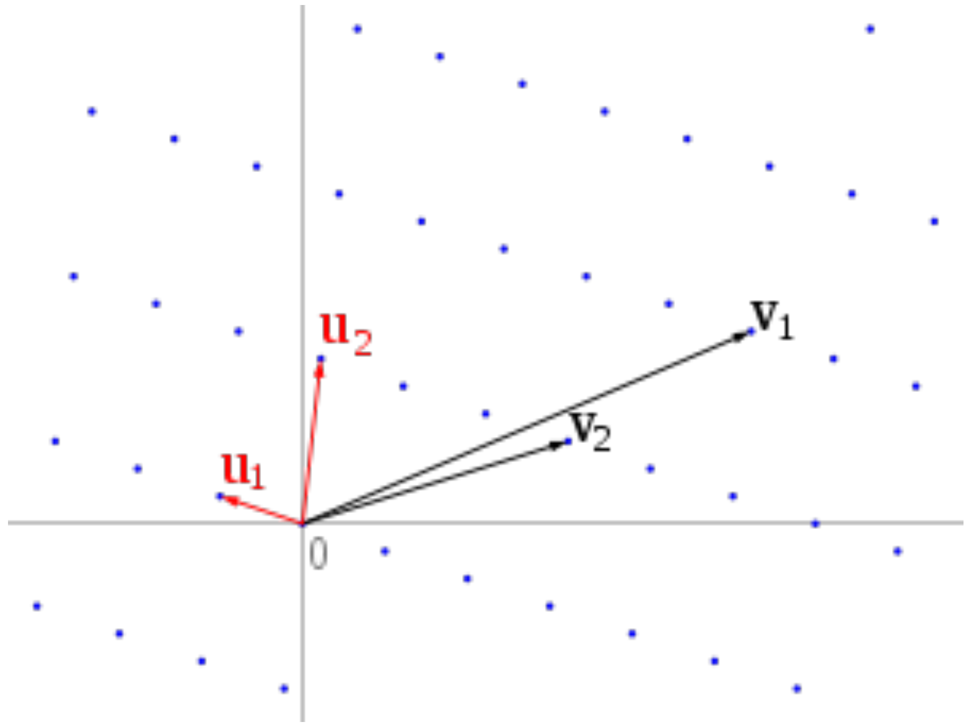
- <u>Decryption algorithm Dec</u>, is such that

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

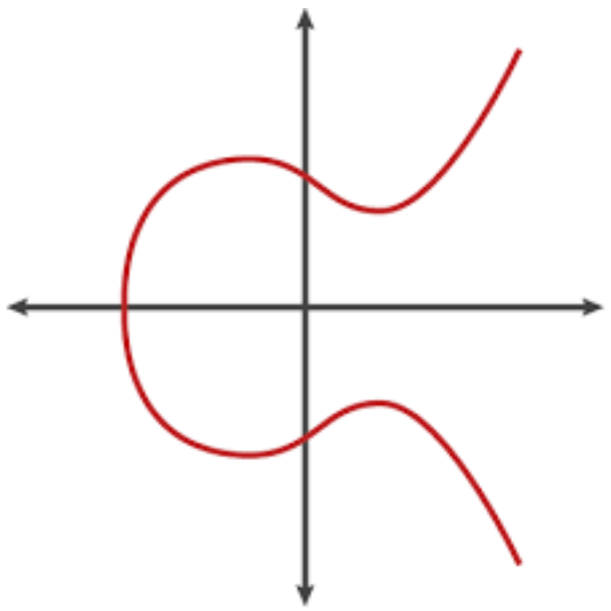
Public-Key Encryption in Action



All known Public-Key Encryption uses…



MATH



$$N = pq$$

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

Modular Arithmetic: Two sets

$$\mathbb{Z}_N = \{0,1,\ldots,N-1\}$$

$$\mathbb{Z}_N^* = \{i : \gcd(i,N) = 1\} \quad (\mathbb{Z}_N^* \subsetneq \mathbb{Z}_N)$$

gcd = "greatest common divisor"

Examples:

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

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

Defintion: $\phi(N) = |\mathbb{Z}_N^*|$

$$\phi(13) = 12 \quad \phi(15) = 8$$

Modular Arithmetic

Definition

$x \mod N$ means the remainder when x is divided by N.

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

$$2 \times 4 = 8 \mod 15 \qquad 13 \times 8 = 14 \mod 15$$

Theorem:

$\mathbb{Z}_N^{*}$ is “closed under multiplication modulo N”.

RSA “Trapdoor Function”

Lemma: Suppose $e, d \in \mathbb{Z}^*_{\phi(N)}$ satisfy $ed = 1 \mod \phi(N)$. Then for any $x \in \mathbb{Z}_N$ we have that

$$(x^e)^d = x^{ed} = x \mod N$$

Example: $N = 15,\ \phi(N) = 8,\ e = 3,\ d = 3$

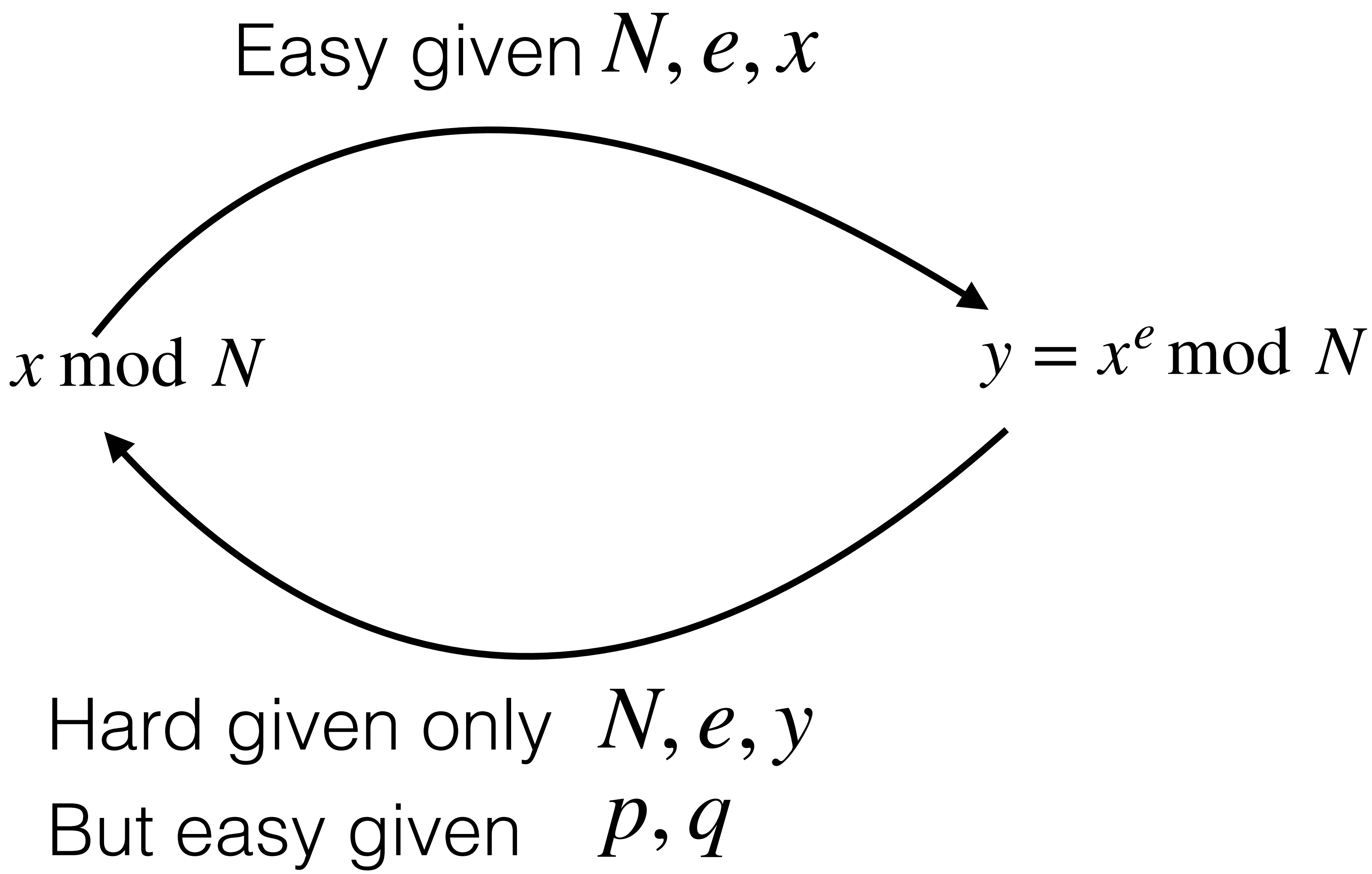
The satisfy condition in lemma: $ed = 3 \cdot 3 = 9 = 1 \mod 8$

So “powering by 3” always un-does itself.

$$(5^3)^3 = 5^9 = 1953125 = 5 \mod 15$$

Usually e and d are different.

RSA "Trapdoor Function"



Finding "e-th roots modulo N" is hard.
Contrast is usual arithmetic, where finding roots is easy.

RSA "Trapdoor Function"

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

$$\text{Enc}((N, e), M) = M^e \bmod N$$

$$\text{Dec}((N, d), C) = C^d \bmod N$$

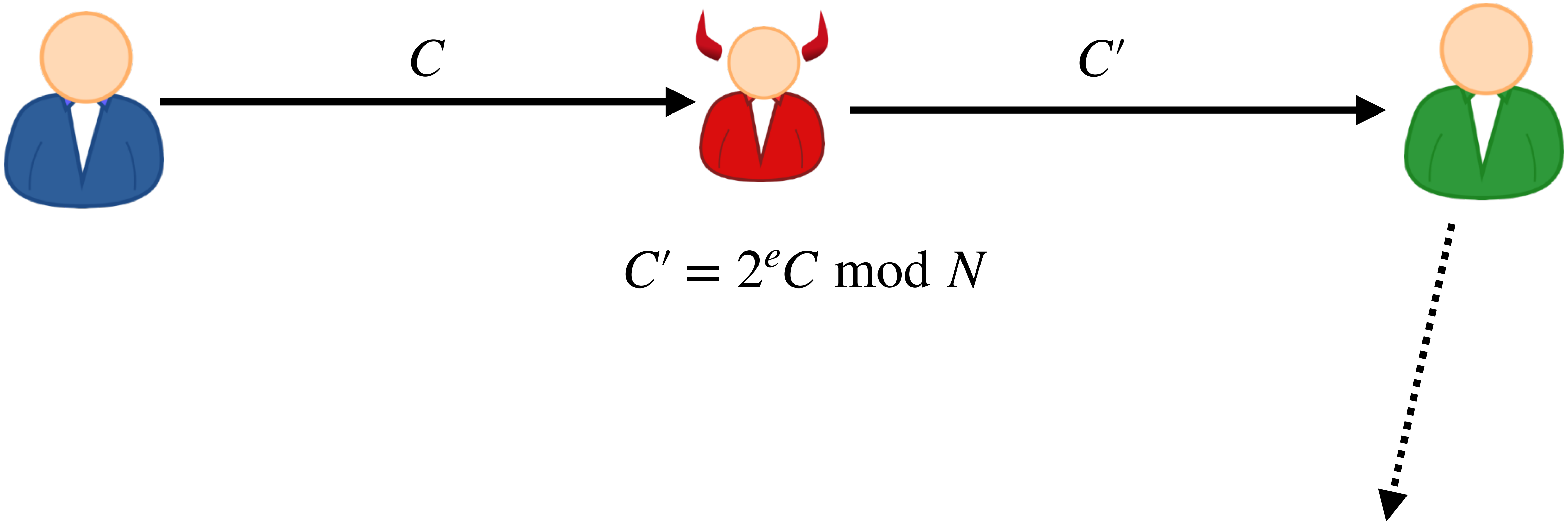
Messages and ciphertexts are in $\mathbb{Z}_N^*$

Setting up RSA:

- Need two large random primes
- Have to pick e and then find d
- Don't worry about how exactly

Non-Integrity of the RSA Trapdoor Function

$\text{Enc}((N, e), M) = M^e \bmod N = C$



$C' = 2^e C \bmod N$

$(C')^d = (2^e M^e)^d = (2M)^{ed} = 2M \bmod N$

Encryption with the RSA Trapdoor Function?

$$\text{Enc}((N, e), M) = M^e \bmod N$$

$$\text{Dec}((N, d), C) = C^d \bmod N$$

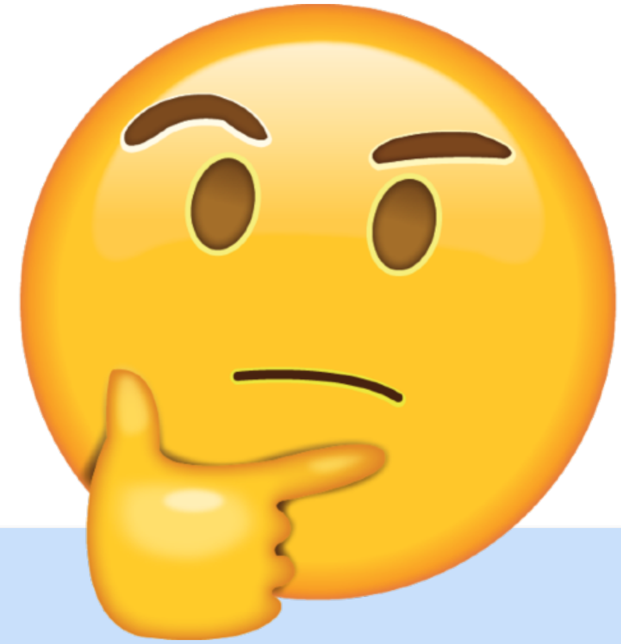
Messages and ciphertexts are in $\mathbb{Z}_N^*$

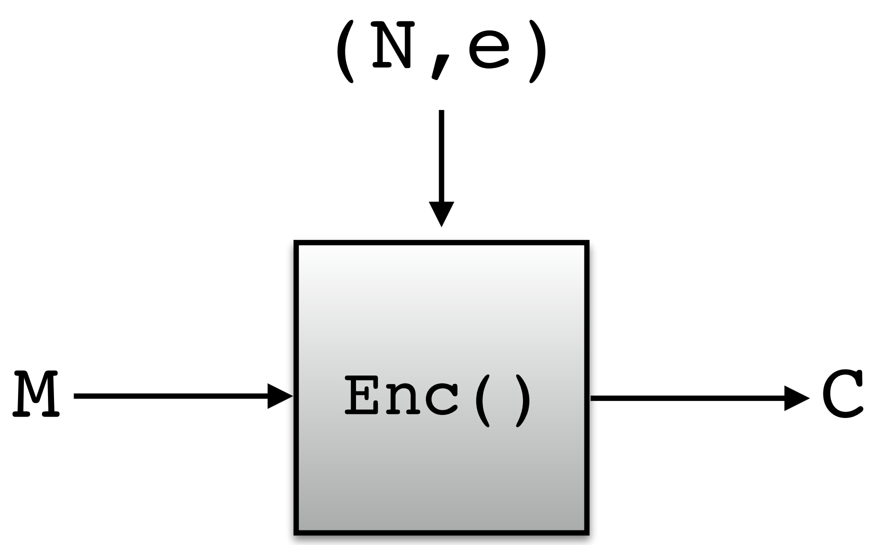
- Several problems
  - Encryption of 1 is 1
  - e=3 is popular. Encryption of 2 is 8… (no wrapping mod N)
  - RSA Trapdoor Function is deterministic

Solution: Pad input M using random (structured) bits.
- Serves purpose of padding **and** nonce/IV randomization

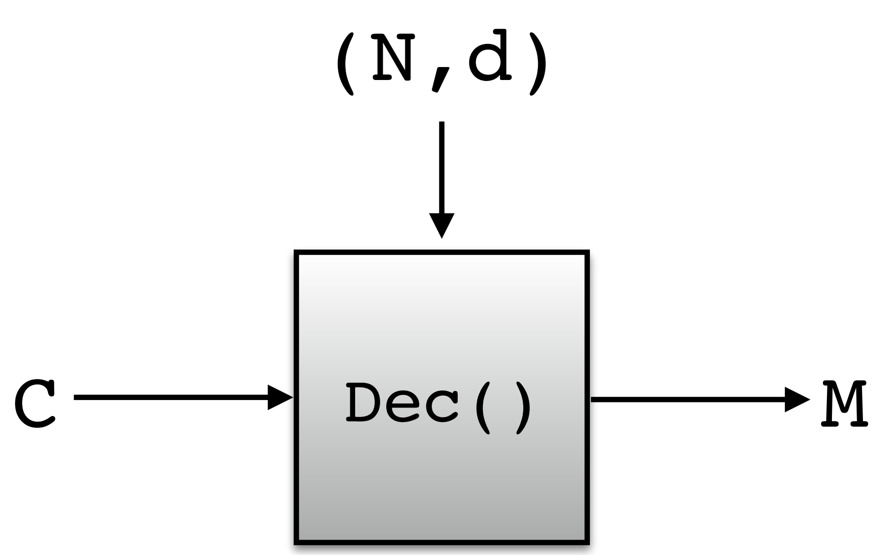
PKCS#1 v1.5 RSA Encryption

N: **n**-byte long integer.
Want to encrypt **m**-byte messages.






```
Enc((N,e),M):
1. pad ← (n-m-3) random non-zero bytes.
2. X←00||02||pad||00||M
3. Output Xe mod N
```




```
Dec((N,d),M):
1. X← Cd mod N
2. Parse X = aa||bb||rest
3. If aa≠00 or bb≠02 or 00∉rest:
       Output ERROR
4. Parse rest = pad||00||M
5. Return M
```

Warning: Broken

Bleichenbacher’s Padding Oracle Attack (1998)

PK=(N,e)

Want to decrypt C

C’ →

← ACCEPT or REJECT

System
(e.g. webserver)
SK=(N,d)

Infer something about
$(C')^d \bmod N$

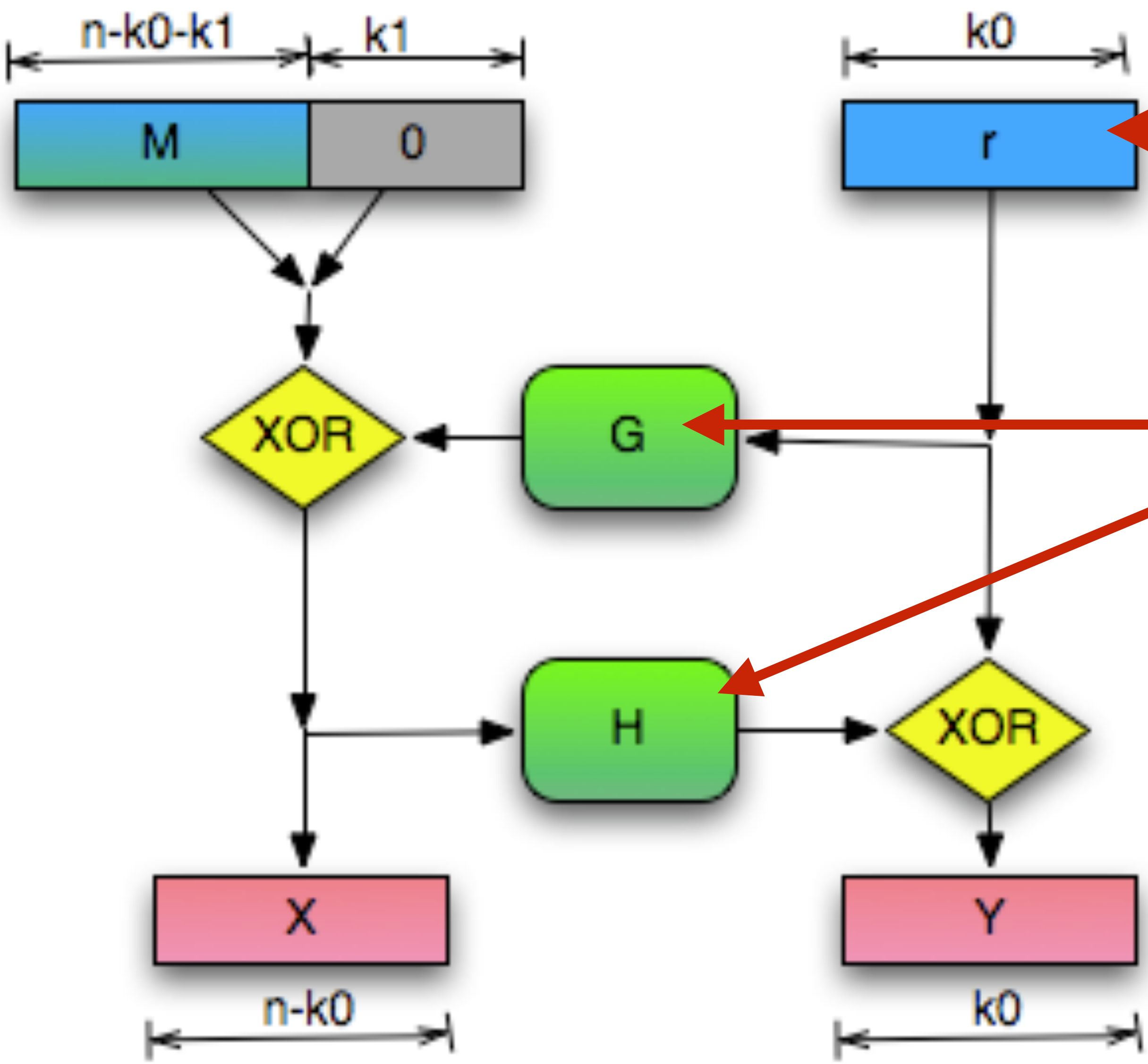
Info about X →

```
Dec((N,d),M):
 1. X← Cd mod N
 2. Parse X = aa||bb||rest
 3. If aa≠00 or bb≠02 or 00∉rest:
        Output ERROR
 4. Parse rest = pad||00||M
 5. Return M
```

Originally needed millions of C’.
Best currently about 10,000.

Better Padding: RSA-OAEP

RSA-OAEP [Bellare and Rogaway, '94] prevents padding-oracle attacks with better padding using a hash function.



random bytes

functions based on hash functions

Uses "Feistel Network" (!)

(Then apply RSA trapdoor function.)

Security of RSA Trapdoor Function Against Inversion

Inverting RSA Trapdoor Function — Given N, e, y find x such that $x^e = y \bmod N$

↑

If we know d… — Compute $x = y^d \bmod N$

↑

If we know $\varphi(N)$… — Compute $d = e^{-1} \bmod \varphi(N)$

↑

If we know p, q… — Compute $\varphi(N) = (p-1)(q-1)$

↑

But if we only know N… — Learning p and q from N is called the *factoring problem*.

- In principle one may invert RSA without factoring N, but it is the only approach known.

Naive Factoring Algorithm

- Given input `N=901`, what are `p`,`q`?

```
NaiveFactor(N):

  1. For i=2…sqrt(N):
       If i divides N:
         Output p=i, q=N/i
```

- Runtime is `sqrt(N)`≪`N`
- But `sqrt(N)` is still huge (e.g. $\mathtt{sqrt}(2^{2048})=2^{1024}$)

Factoring Algorithms

- If we can factor **N**, we can find **d** and break any version of RSA.

Algorithm	Time to Factor N
Naive: Try dividing by 1,2,3,…	$O(N^{.5}) = O(e^{.5\ln(N)})$
Quadratic Sieve	$O(e^{c})$ $c = (\ln N)^{1/2}(\ln\ln N)^{1/2}$
Number Field Sieve	$O(e^{c})$ $c = 1.9(\ln N)^{1/3}(\ln\ln N)^{2/3}$

- Total break requires $c = O(\ln\ln N)$

Factoring Records

- 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

Public-Key Encryption in Practice: Hybrid Encryption

- RSA runs reasonably fast but is orders of magnitude slower than symmetric encryption with AES.
  - My laptop…
    - Can encrypt 800 MB per second using AES-CBC
    - Can only evaluate RSA 1000 times per second

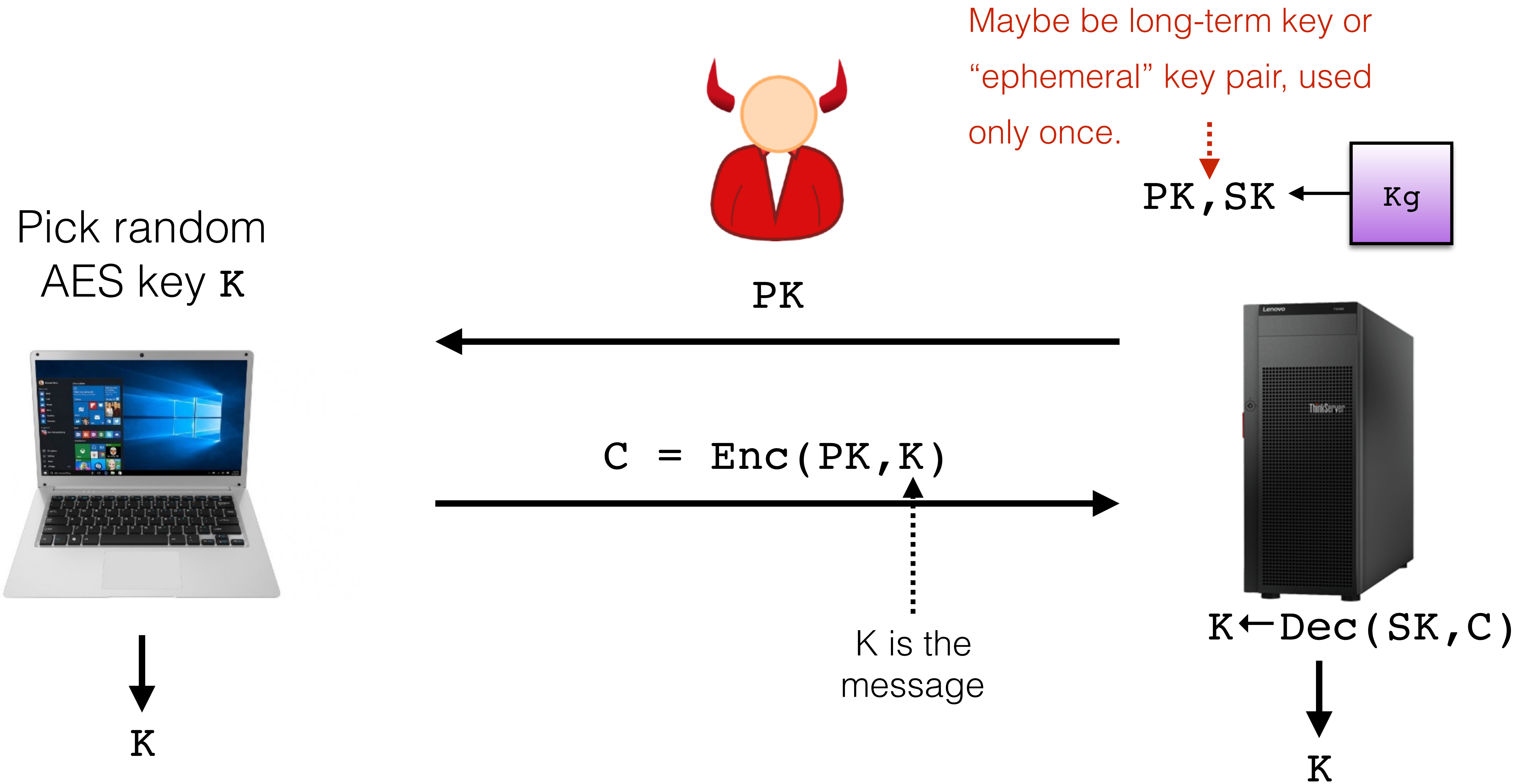
Solution: Use public-key encryption to send a 16-byte key K for AES. Then encrypt rest of traffic using authenticated encryption.

- Called "hybrid encryption"

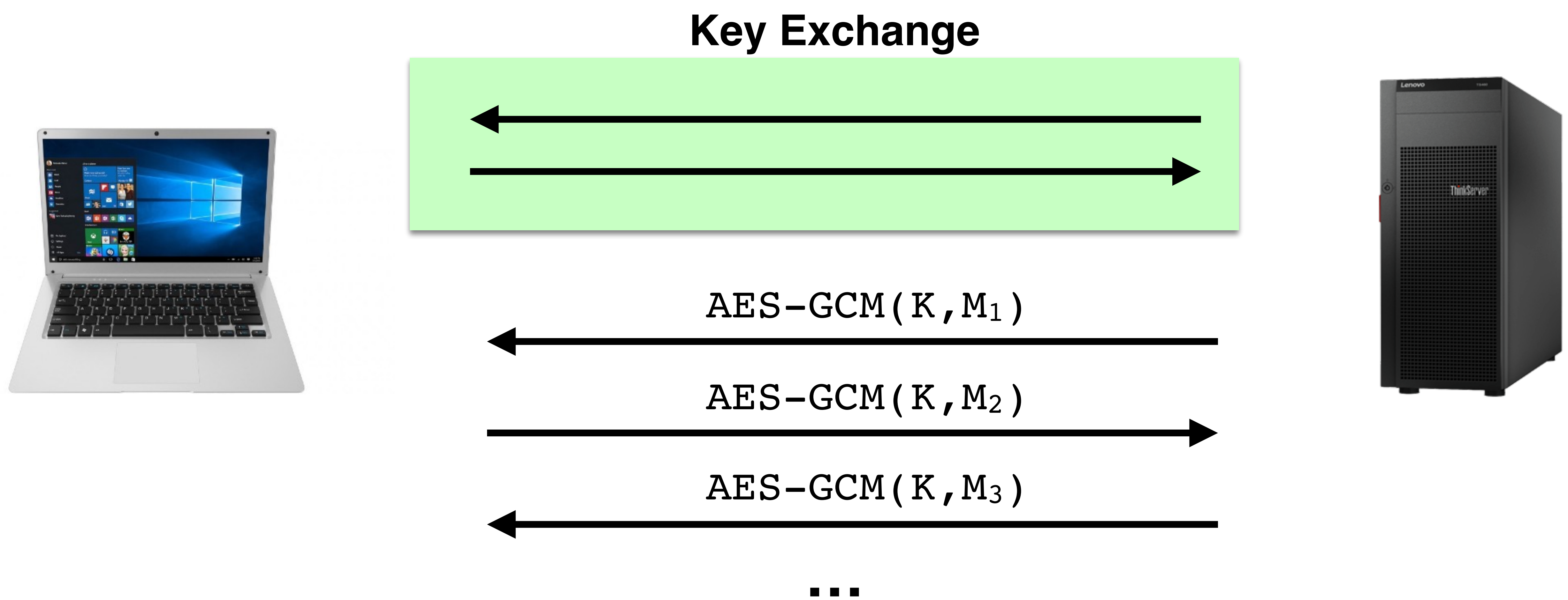
Key Exchange and Hybrid Encryption

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

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



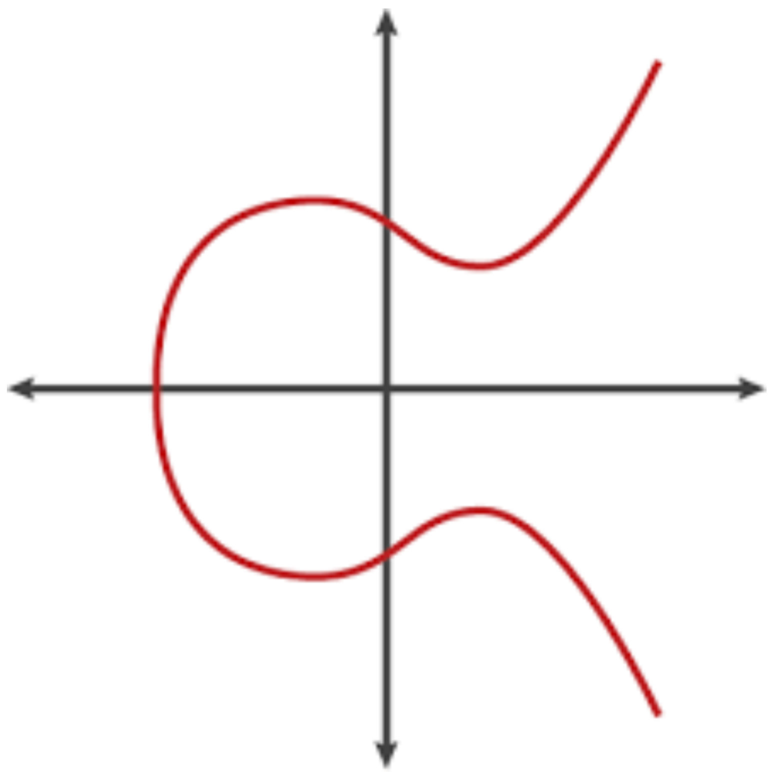
Key Exchange and Hybrid Encryption

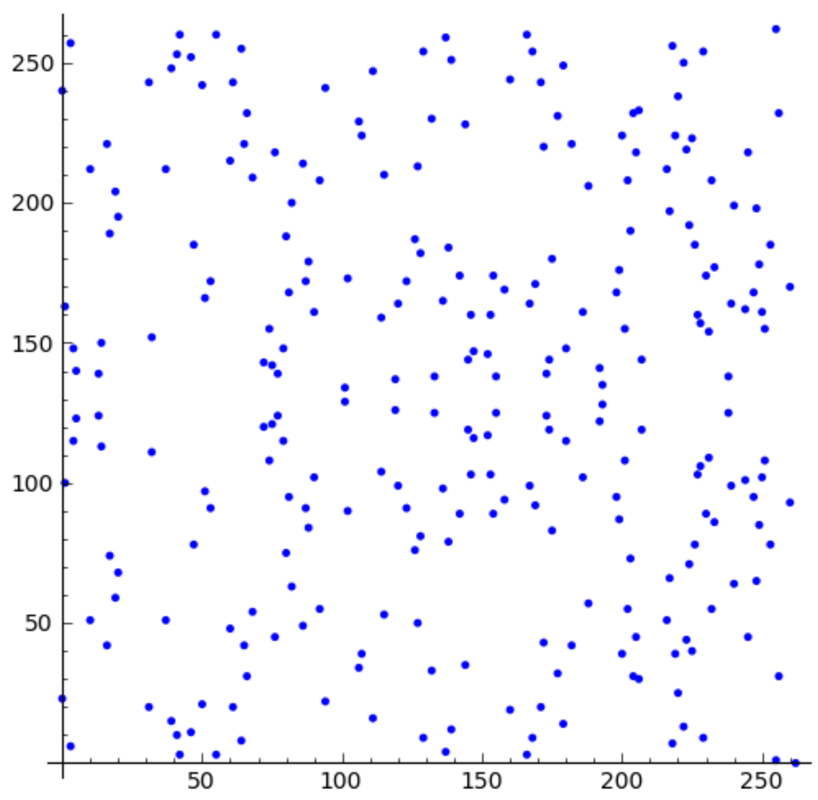


- After up-front cost, bulk encryption is very cheap
- TLS/SSH (covered later) Terminology:
  - “Handshake” = key exchange
  - “Record protocol” = symmetric encryption phase

Key Exchange Going Forward: Elliptic Curve Diffie-Hellman

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





- Will be covered when we do secure messaging!

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.

Huge quantum computers will break:

- RSA (any padding)
- Diffie-Hellman

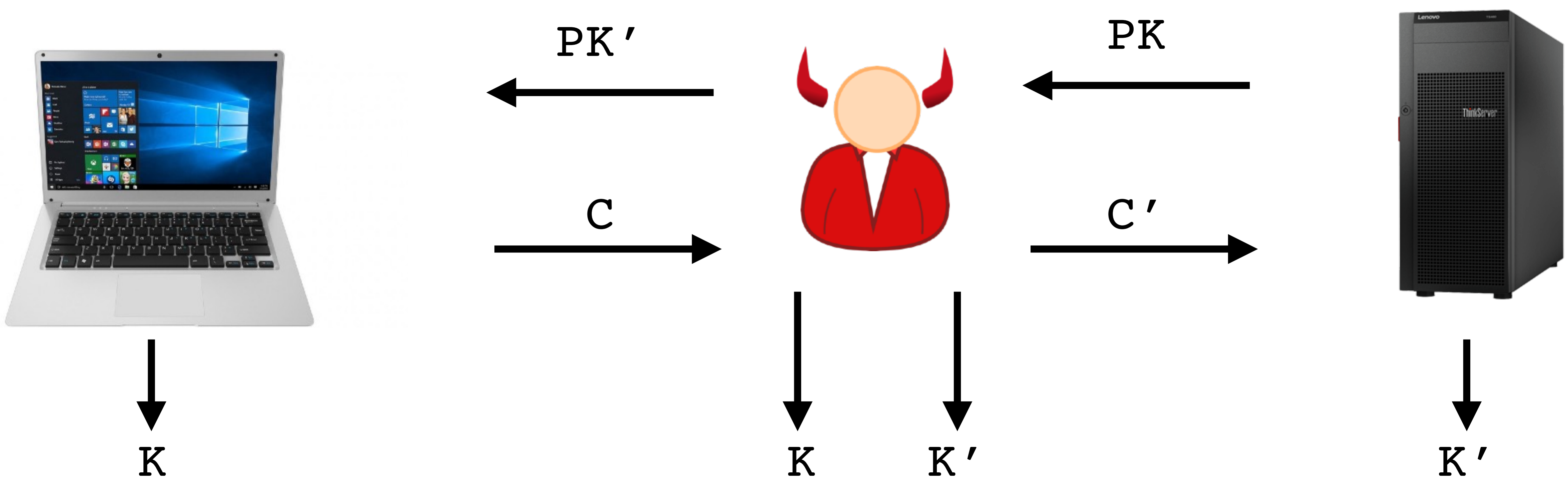
Shor's algorithm, 1994

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

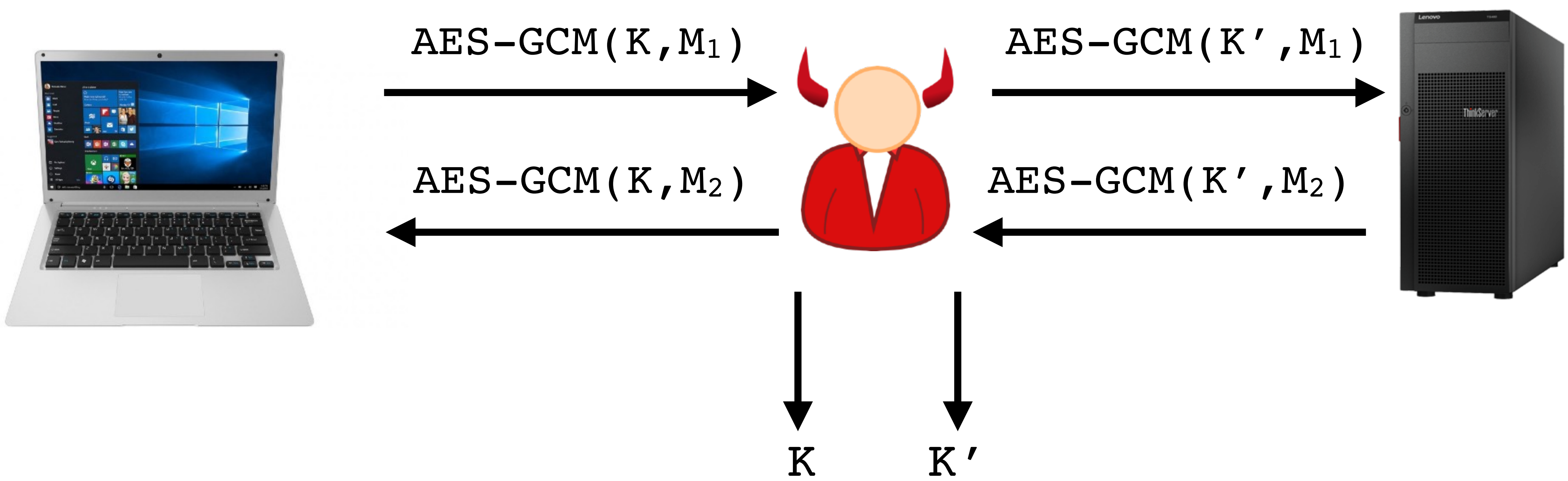
Key Exchange with a Person-in-the-Middle

Adversary may silently sit between parties and modify messages.



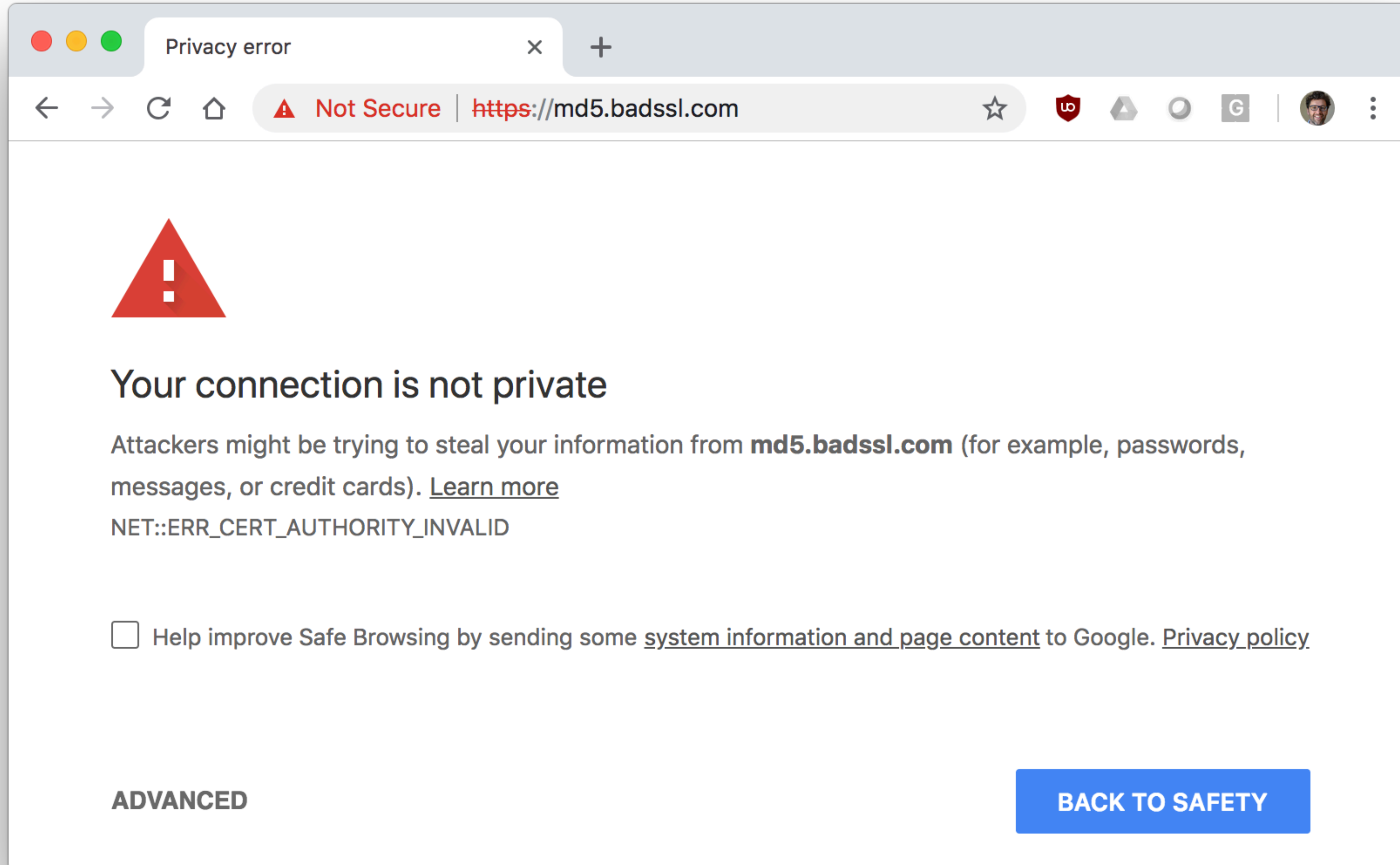
Parties agree on different keys, both known to adversary…

Key Exchange with a Person-in-the-Middle



Connection is totally transparent to adversary.

Translation is invisible to parties.


Privacy error
Not Secure
https://md5.badssl.com
Your connection is not private
Attackers might be trying to steal your information from md5.badssl.com (for example, passwords, messages, or credit cards). Learn more
NET::ERR_CERT_AUTHORITY_INVALID
Help improve Safe Browsing by sending some system information and page content to Google. Privacy policy
ADVANCED
BACK TO SAFETY

Next up: Tool for Stopping the Person-in-the-Middle

- Digital Signatures

Later during networking week:

- Public-Key Infrastructure (PKI)
- Certificates and chains of trust

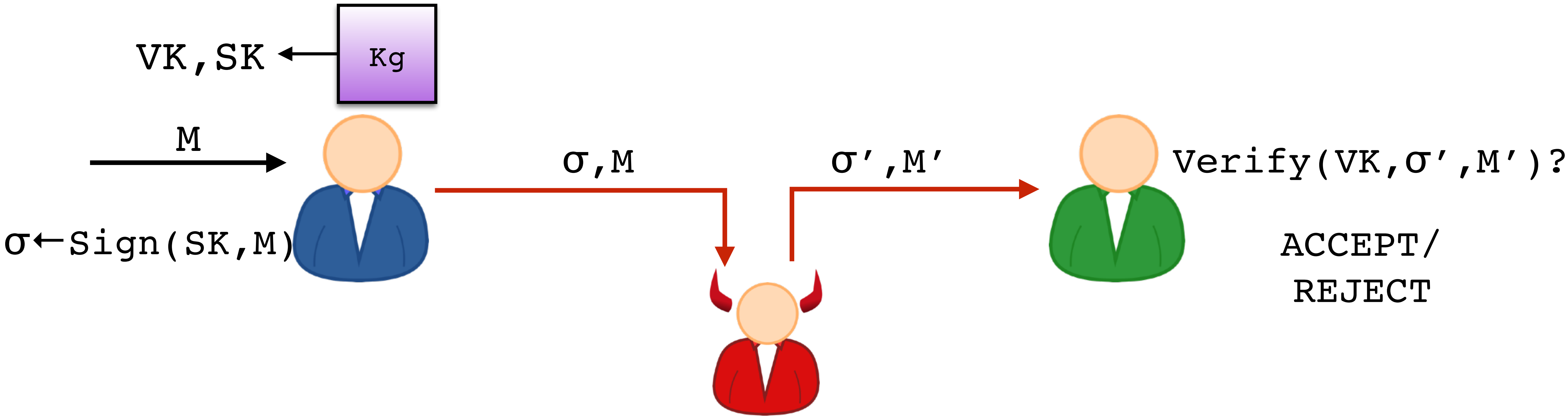
Crypto Tool: Digital Signatures

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

- <u>Key generation algorithm **`Kg`**</u>, takes no input and outputs a (random) public-verification-key/secret-signing key pair **`(VK,SK)`**
- <u>Signing algorithm **`Sign`**</u>, takes input the secret key **`SK`** and a message **`M`**, outputs "signature" **σ←`Sign(SK,M)`**
- <u>Verification algorithm **`Verify`**</u>, takes input the public key V`K`, a message **`M`**, a signature **σ**, and outputs `ACCEPT/REJECT`

`Verify(VK,M,σ)=ACCEPT/REJECT`

Digital Signature Security Goal: Unforgeability



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

"Plain" RSA with No Encoding

Broken

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

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

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

Messages & sigs are in $\mathbb{Z}_N^*$

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

“Plain” RSA Weaknesses

Broken

Assume `e=3`.

$$\mathrm{Sign}((N,d),M) = M^d \bmod N \qquad \mathrm{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message M': Find number σ' such that $(\sigma')^3 = M' \bmod N$

M=1 weakness: If $M'=1$ then it is easy to forge. Take $\sigma'=1$:

$$(\sigma'^3) = 1^3 = 1 = M' \bmod N \quad ✓$$

Cube-M weakness: If M' is a *perfect cube* then it is easy to forge.
Just take $\sigma' = (M')^{1/3}$:, i.e. the usual cube root of M':

Example: To forge on $M'=8$, which is a perfect cube, set $\sigma'=2$.

$$(\sigma')^3 = 2^3 = 8 = M' \bmod N \quad ✓$$

(Intuition: If cubing does not “wrap modulo N”, then it is easy to un-do.)

Further "Plain" RSA Weaknesses

Broken

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message `M'`: Find number `σ'` such that $(\sigma')^3 = M' \bmod N$

***Malleability* weakness:** If σ is a valid signature for `M`, then it is easy to forge a signature on `8M mod N`.

Given `(M,σ)`, compute forgery `(M',σ')` as

$$M' = (8*M \bmod N), \text{ and } \sigma' = (2*\sigma \bmod N)$$

Then `Verify((N,3),M',σ')` checks:

$$(\sigma')^3 = (2*\sigma \bmod N)^3 = (2^3*\sigma^3 \bmod N) = (2^3*M \bmod N) = 8M \bmod N$$

$\sigma^3 = M \bmod N$ b/c σ is valid sig. on `M`

Further “Plain” RSA Weaknesses

Broken

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message `M’`: Find number `σ’` such that `(σ’)`^{3}`=M’ mod N`

***Backwards signing* weakness:** Generate *some* valid signature by picking `σ’` first, and then defining `M’=(σ’`^{3}` mod N)`

Then `Verify((N,3),M’,σ’)` checks:

`(σ’)`^{3}`=(M’ mod N)` ✓

Further "Plain" RSA Weaknesses

Broken

$$\text{Sign}((N,d),M) = M^d \bmod N \qquad \text{Verify}((N,3),M,\sigma) : \sigma^3 = M \bmod N?$$

To forge a signature on message `M'`: Find number `σ'` such that `(σ')³=M' mod N`

Summary:

- Plain RSA Signatures allow several types of forgeries
- It was sometimes argued that these forgeries aren't important: If `M` is english text, then `M'` is unlikely to be meaningful for these attacks
- But often they are damaging anyway

RSA Signatures with Encoding

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

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

Messages & sigs are in $\mathbb{Z}_N^*$

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

encode maps bit strings to numbers in $\mathbb{Z}_N^*$

Encoding needs to address:

- Small M or M = perfect cube
- Malleability
- Backwards signing

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

☣ Broken ☣

RSA Signature Padding: PKCS #1 v1.5

Note: We already saw PKCS#1 v1.5 *encryption* padding. This is *signature* padding. It is different.

`N`: `n`-byte long integer.
`H`: Hash function.
`hash_id`: Magic number assigned to H

Ex: for `H=SHA-256`, `hash_id = 3051…0440`

```
Sign((N,d),M):
1. digest←hash_id||H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 'FF' bytes
3. X←00||01||pad||00||digest
4. Output σ = X^d mod N
```

```
Verify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
     or Y≠(FF)^(n-m-3)
     or digest≠hash_id||H(M):
       Output REJECT
4. Else: Output ACCEPT
```

Encoding needs to address:

- Perfect cubes → The high-order bits + digest means X is large and random-looking, rarely a cube.
- Malleability → Stopped by hash, ex: `H(2*M)≠2*H(M)`
- Backwards signing → Stopped by hash: given `digest`, hard to find `M` such that `H(M)=digest`.

RSA Signature Padding: PKCS #1 v1.5

Note: We already saw PKCS#1 v1.5 *encryption* padding. This is *signature* padding. It is different.

`N`: `n`-byte long integer.
`H`: Hash function.
`hash_id`: Magic number assigned to H

Ex: for `H=SHA-256`, `hash_id = 3051…0440`

```
Sign((N,d),M):
1. digest←hash_id||H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 'FF' bytes
3. X←00||01||pad||00||digest
4. Output σ = X^d mod N
```

```
Verify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
   or Y≠(FF)^(n-m-3)
   or digest≠hash_id||H(M):
      Output REJECT
4. Else: Output ACCEPT
```

Introduces new weakness:

- Hash collision attacks: If `H(M) = H(M')`, then …

`Sign((N,d),M) = Sign((N,d),M')`

- i.e., can reuse a signature for `M` as a signature for `M'`

Now: A Buggy Implementation, with an Attack

- Padding check is often implemented incorrectly

- Enables forging of signatures on *arbitrary* messages

Real-world attacks against:

- OpenSSL (2006)
- Apple OSX (2006)
- Apache (2006)
- VMWare (2006)
- All the biggest Linux distros (2006)
- Firefox/Thunderbird (2013)

…

(at least 6 more in 2018 alone)

https://i.blackhat.com/USA-19/Wednesday/us-19-Chau-A-Decade-After-Bleichenbacher-06-RSA-Signature-Forgery-Still-Works-wp.pdf

Broken

Buggy Verification in PKCS #1 v1.5 RSA Signatures

```
Sign((N,d),M):
1. digest←hash_id||H(M) // m bytes long
2. pad←FF||FF||…||FF// n-m-3 'FF' bytes
3. X←00||01||pad||00||digest
4. Output σ = X^d mod N
```

```
Verify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
     or Y≠(FF)^(n-m-3)
     or digest≠hash_id||H(M):
       Output REJECT
4. Else: Output ACCEPT
```

```
BuggyVerify((N,3),M,σ):
1. X←(σ^3 mod N)
2. Parse X→aa||bb||rest
3. If aa≠00 or bb≠01:
      Output REJECT
4. Parse rest=(FF)^p||00||digest||…,
     where p is any positive number
5. If digest≠hash_id||H(M):
      Output REJECT
6. Else: Output ACCEPT
```

Checks if **rest** starts with any number of **FF** bytes followed by a **00** byte.

If so, it takes the next **m** bytes as **digest**.

Correct: X = 00 01 FF FF FF FF FF FF FF FF 00 <DIGEST>

Buggy: X = 00 01 FF 00 <DIGEST> <IGNORED ………… BYTES>

One or more **FF** bytes

Broken

Attacking Buggy Verification

One or more **FF** bytes

Buggy: X = 00 01 FF 00 <DIGEST> <IGNORED ………… BYTES>

To forge a signature on message **M'**: Find number σ' such that

$(\sigma')^3$= 00 01 FF 00 H(M') <JUNK> mod N

We'll use one **FF** byte

m bytes long

n-m-4 bytes free for attacker to pick

Freedom to pick <JUNK> means we can take any σ' such that:

00 01 FF 00 H(M') 00 …… 00 ≤ $(\sigma')^3$ ≤ 00 01 FF 00 H(M') FF …… FF

Sufficient to find: Any perfect cube in the given range. Then apply perfect cube attack.

Fun! (Assignment 2)

Steps in Attack

1. Pick **`M`** you want to forge on
2. Compute lower and upper bounds (numbers), using **`H(M).`**
3. Find a perfect cube **`x`** within allowed range
4. Output cube root of **`x`** as forged signature **σ**.

Attack Summary

- When padding check allows variable number of `FF` bytes, forging is easy
  - Only requires a simple search for a perfect cube in a given range
- *Why did so many make this error?*
  - I don't *really* know for sure
  - My guesses:
    - Plugging in libraries for padding removal without checks.
    - Specifically, ASN.1 parsing libraries are used to remove padding. These are overkill and programmers do not fully understand their behavior (but they also don't want to do the parsing by hand).
    - Traditional unit testing is hard to apply to crypto.
- Attack defeated by using large `e=65537`

Other RSA Padding Schemes: 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 = X^d mod N

Verify((N,e),M,σ):

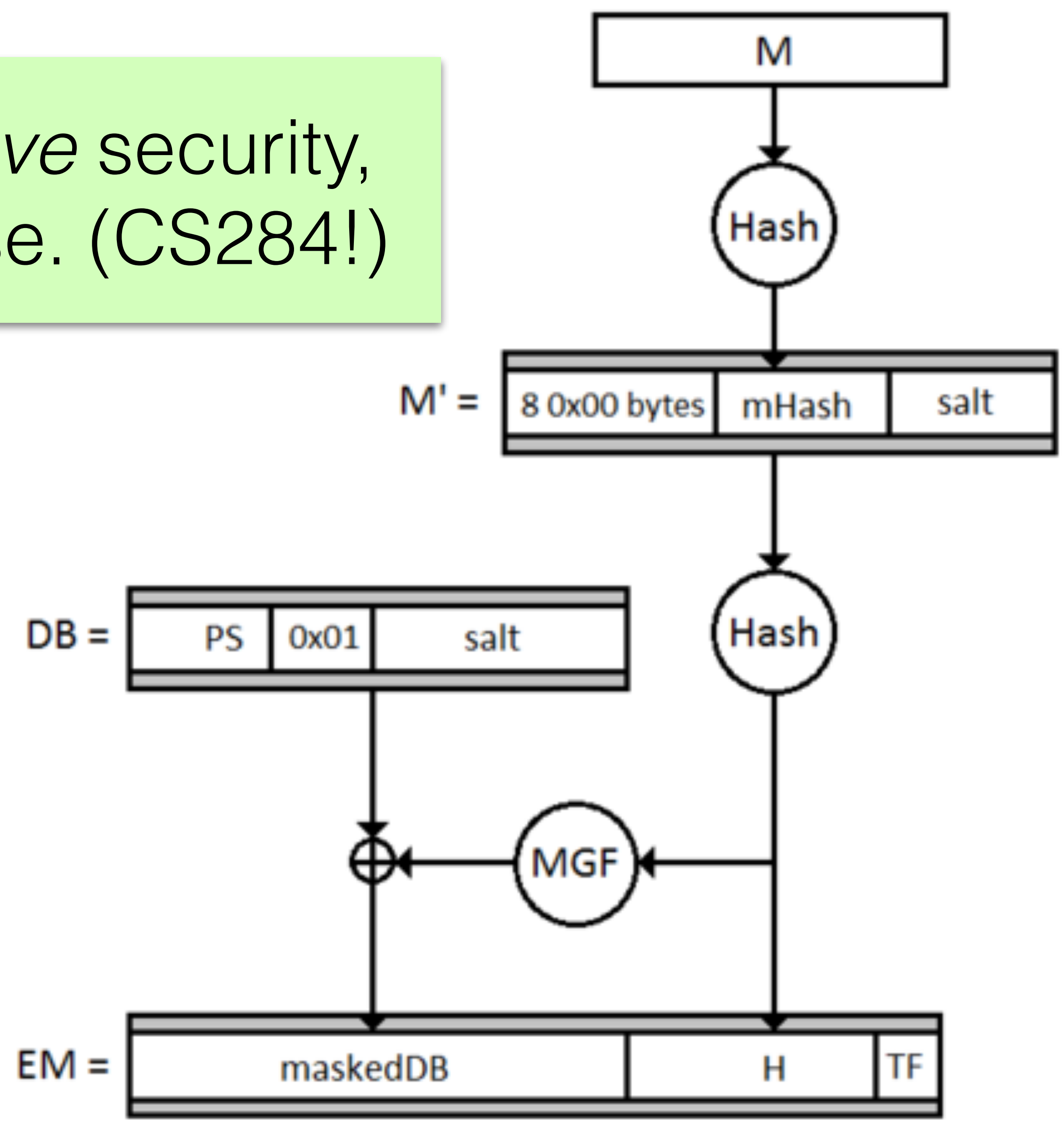
1. X←00||H(1||M)||H(2||M)||…||H(k||M)
2. Check if σ^e = X mod N

Bonus: Can *prove* security, in a strong sense.

Other RSA Padding Schemes: PSS (In TLS 1.3)

- Somewhat complicated
- *Randomized* signing

<u>Bonus:</u> Can *prove* security, in a strong sense. (CS284!)



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

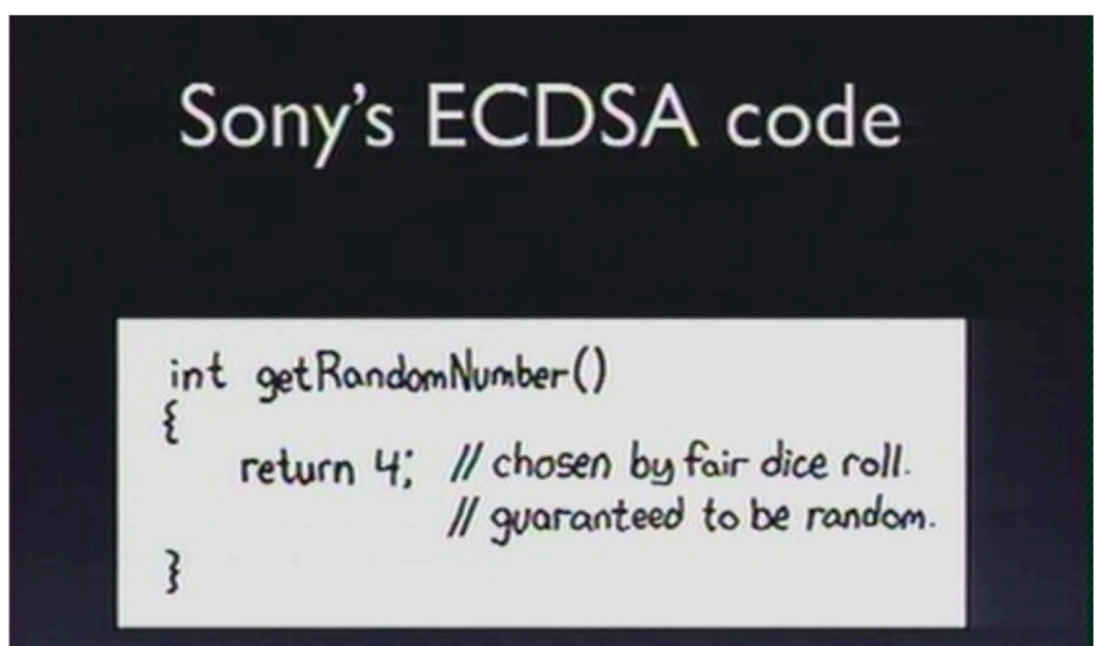
- Based on ideas related to Diffie-Hellman key exchange
- Secure, but ripe for implementation errors

—

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

Sean Hollister
12.29.10

2
Shares



Bonus: New Signature Vulnerability Yesterday!

https://blog.lessonslearned.org/chain-of-fools/

https://media.defense.gov/2020/Jan/14/2002234275/-1/-1/0/CSA-WINDOWS-10-CRYPT-LIB-20190114.PDF

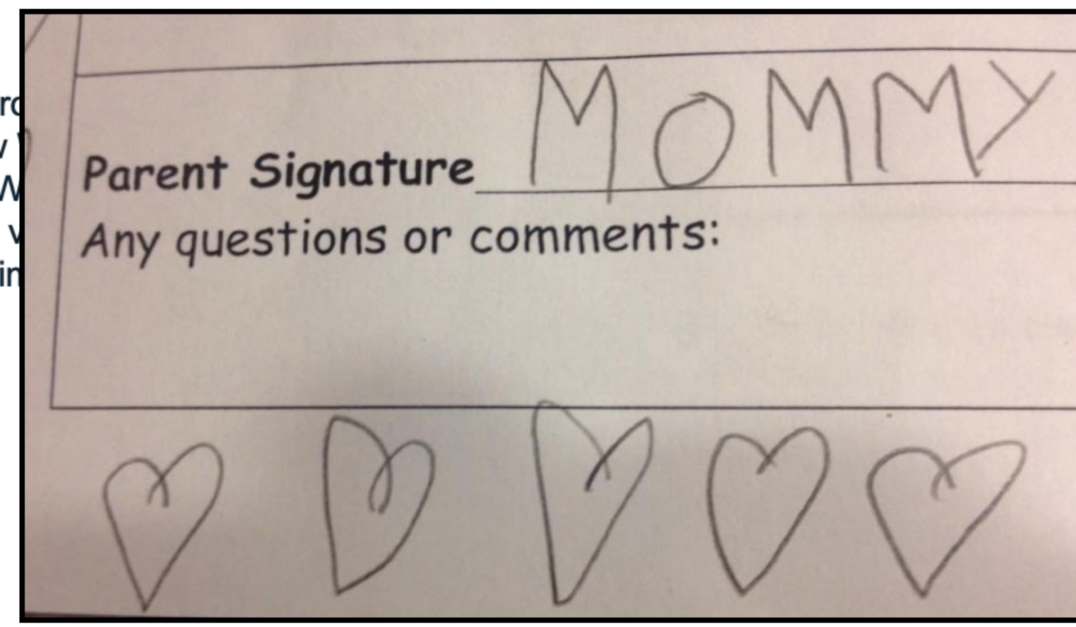
National Security Agency | Cybersecurity Advisory

Patch Critical Cryptographic Vulnerability in Microsoft Windows Clients and Servers

Summary

NSA has discovered a critical vulnerability (CVE-2020-0601) affecting Micro
The certificate validation vulnerability allows an attacker to undermine how
enable remote code execution. The vulnerability affects Windows 10 and W
applications that rely on Windows for trust functionality. Exploitation of the v
network connections and deliver executable code while appearing as legitin
validation of trust may be impacted include:

- HTTPS connections
- Signed files and emails
- Signed executable code launched as user-mode processes



- Details not known yet, but it looks like Windows was not checking crucial parameters before doing signature verification
- Windows was accepting malicious code as authentic.

The End