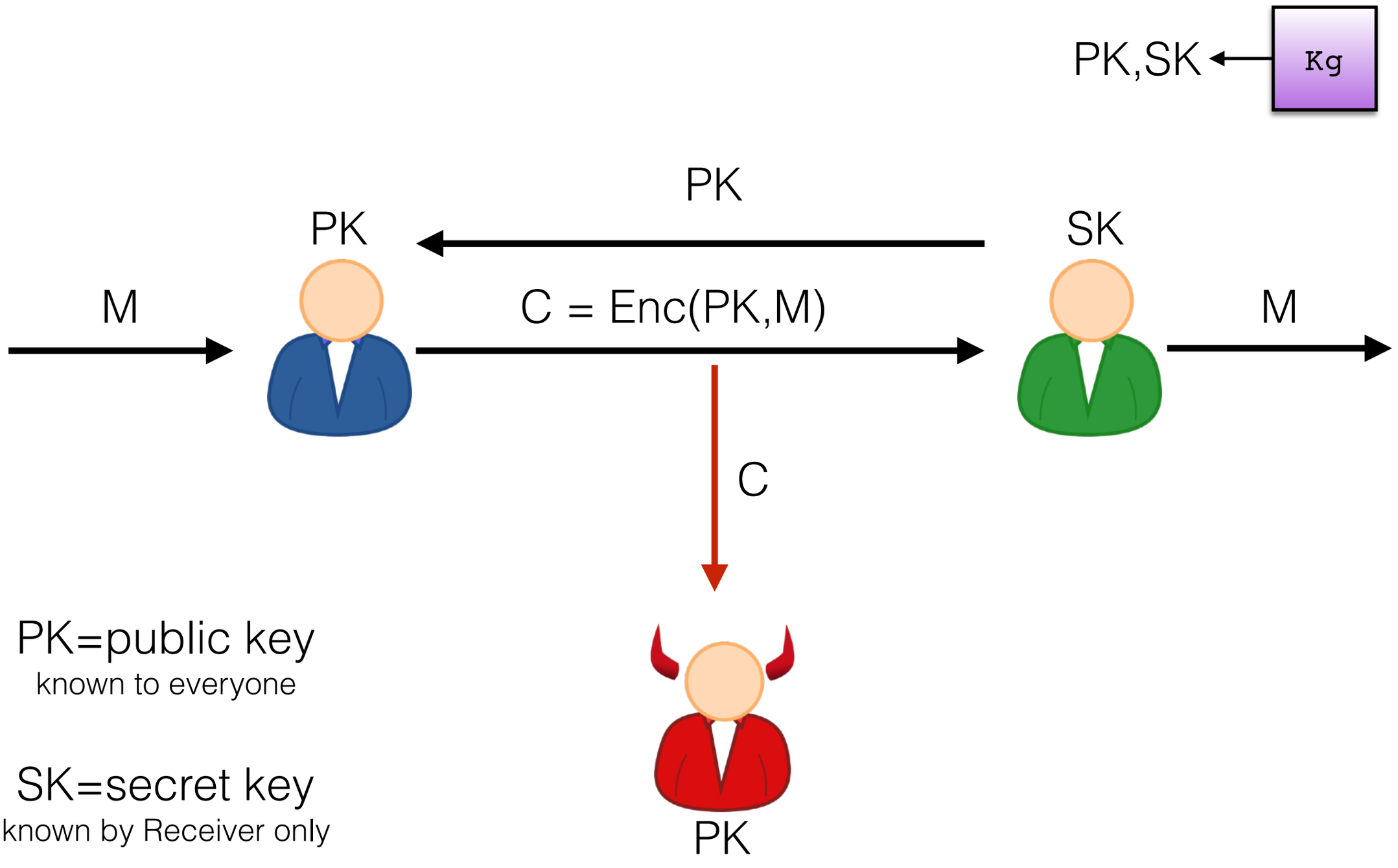
Digital Signatures, Certificates, and TLS

CMSC 23200/33250, Winter 2020, Lecture 6

David Cash and Blase Ur

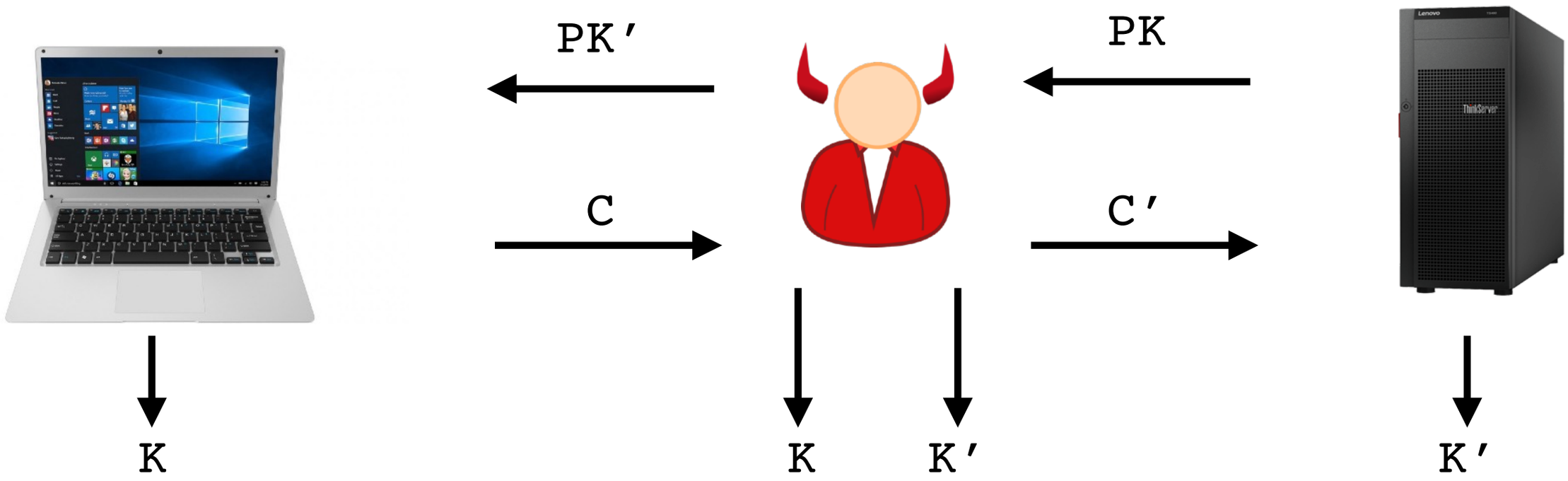
University of Chicago

Public-Key Encryption in Action



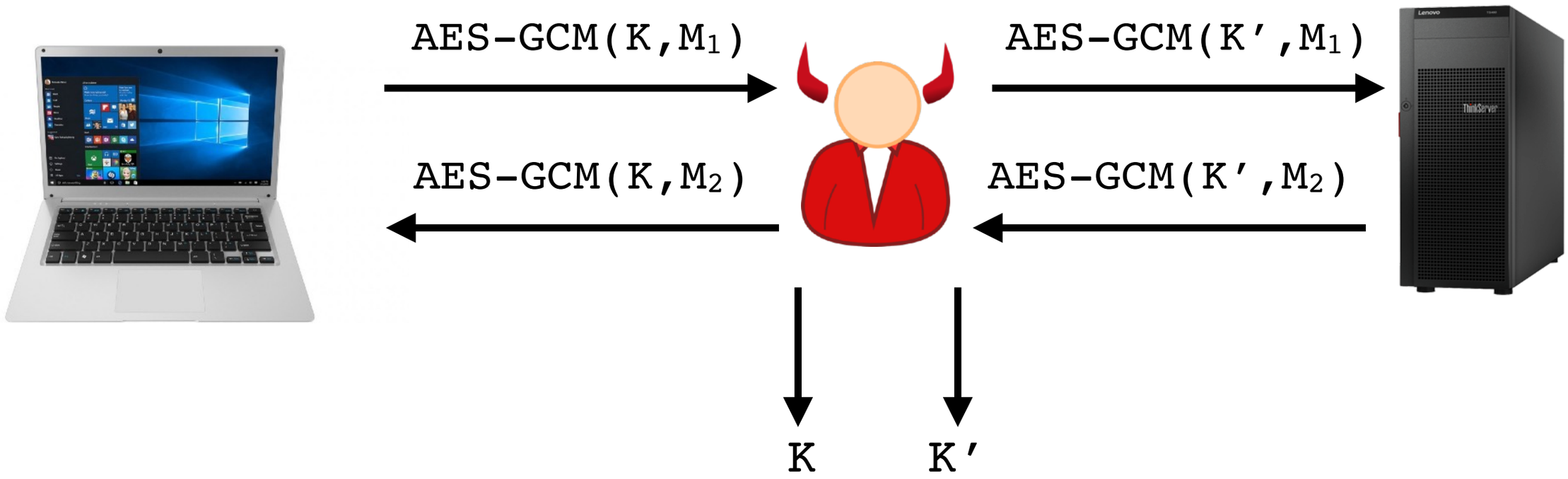
Key Exchange with a Person-in-the-Middle

Adversary may silently sit between parties and modify messages.



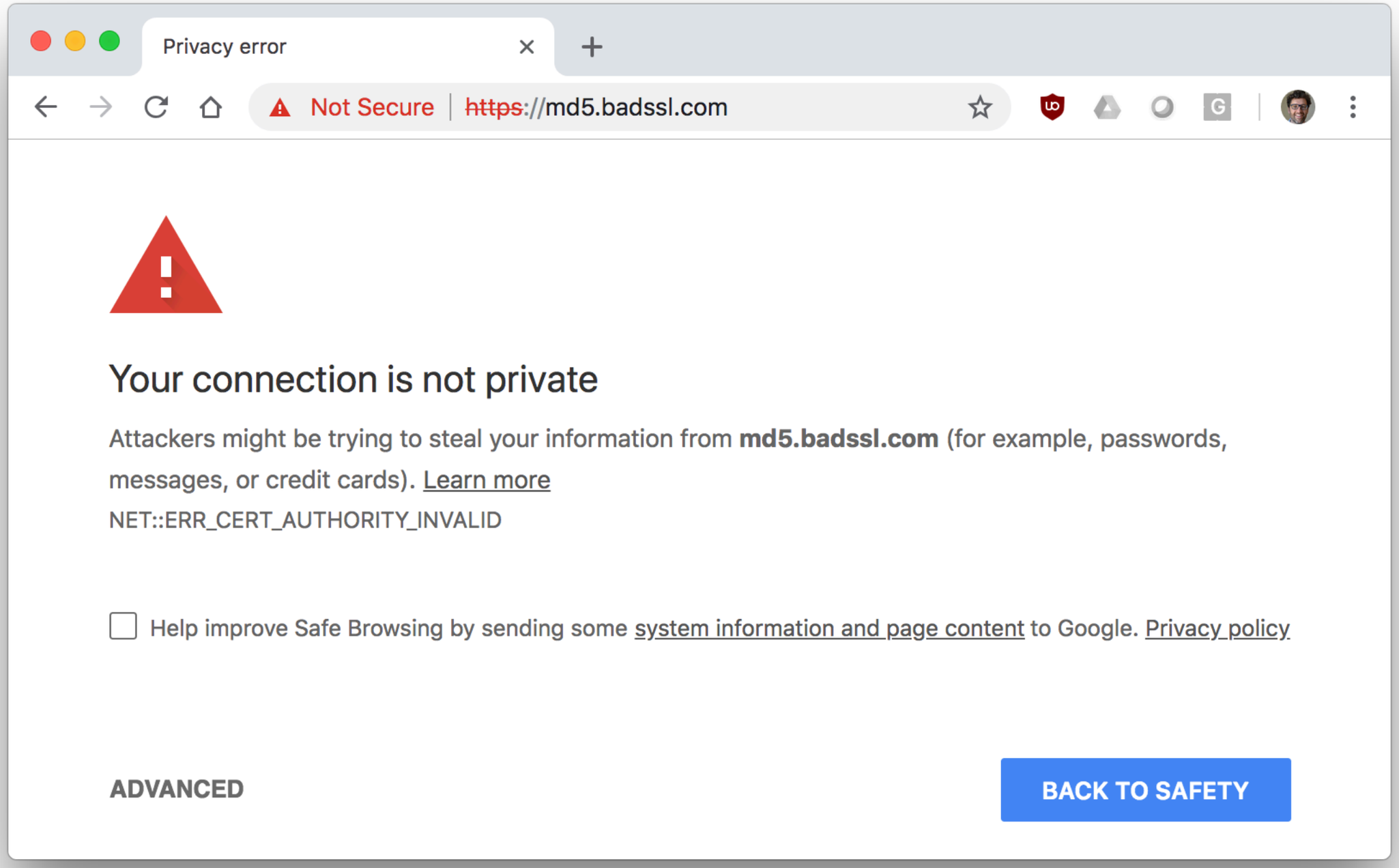
Parties agree on different keys, both known to adversary…

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

https://www.schneier.com/blog/archives/2013/09/new_nsa_leak_sh.html

Blog >

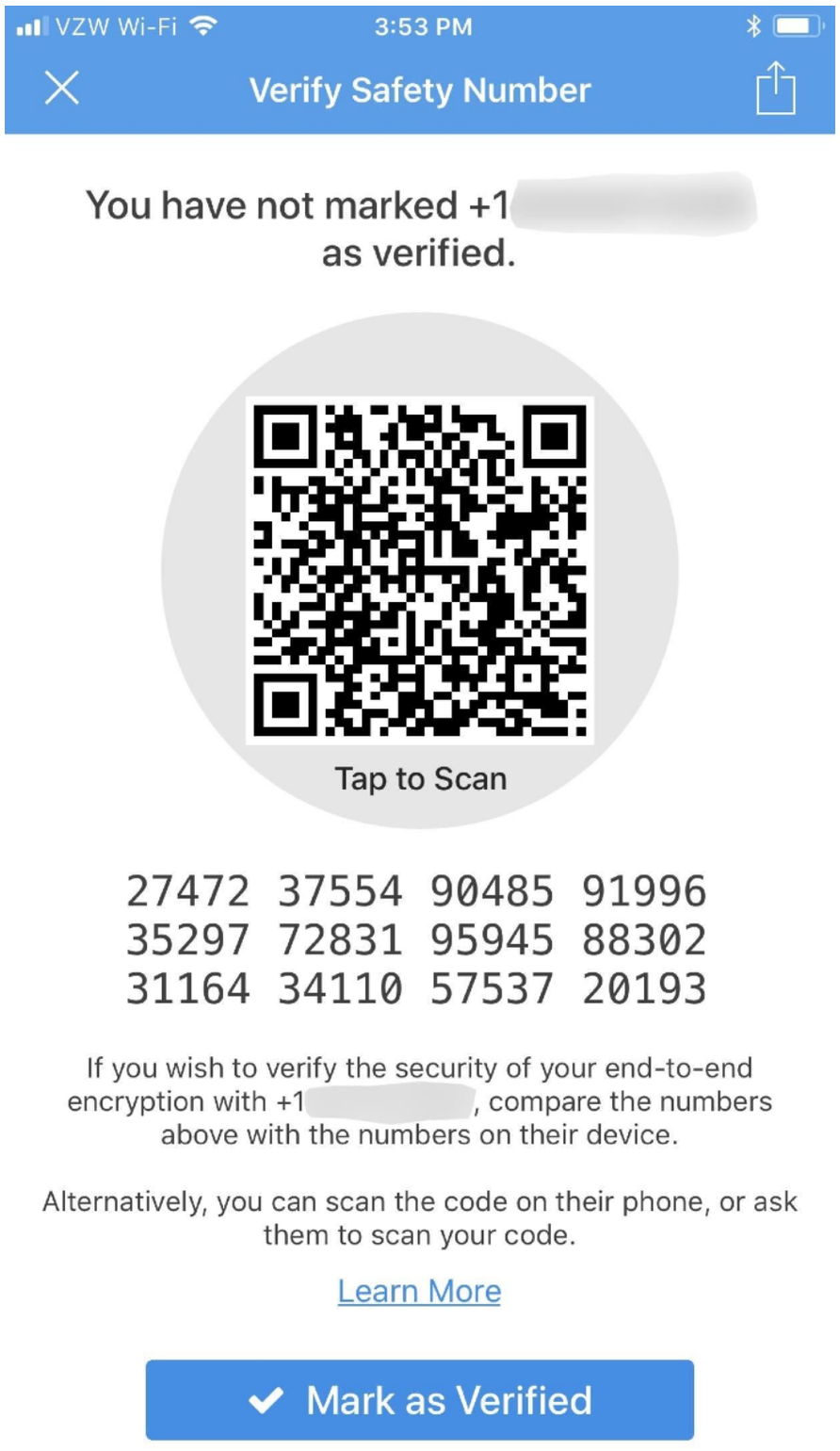
New NSA Leak Shows MITM Attacks Against Major Internet Services

The Brazilian television show "Fantastico" exposed an NSA training presentation that discusses how the agency runs man-in-the-middle attacks on the Internet. The point of the story was that the NSA engages in economic espionage against Petrobras, the Brazilian giant oil company, but I'm more interested in the tactical details.

The video on the webpage is long, and includes what I assume is a dramatization of an NSA classroom, but a few screen shots are important. The pages from the training presentation describe how the NSA's MITM attack works:

> However, in some cases GCHQ and the NSA appear to have taken a more aggressive and controversial route -- on at least one occasion bypassing the need to approach Google directly by performing a man-in-the-middle attack to impersonate Google security certificates. One document published by *Fantastico*, apparently taken from an NSA presentation that also contains some GCHQ slides, describes “how the attack was done” to apparently snoop on SSL traffic. The document illustrates with a diagram how

Authenticating PK “Out of Band”



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

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

Crypto Tool: Digital Signatures

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

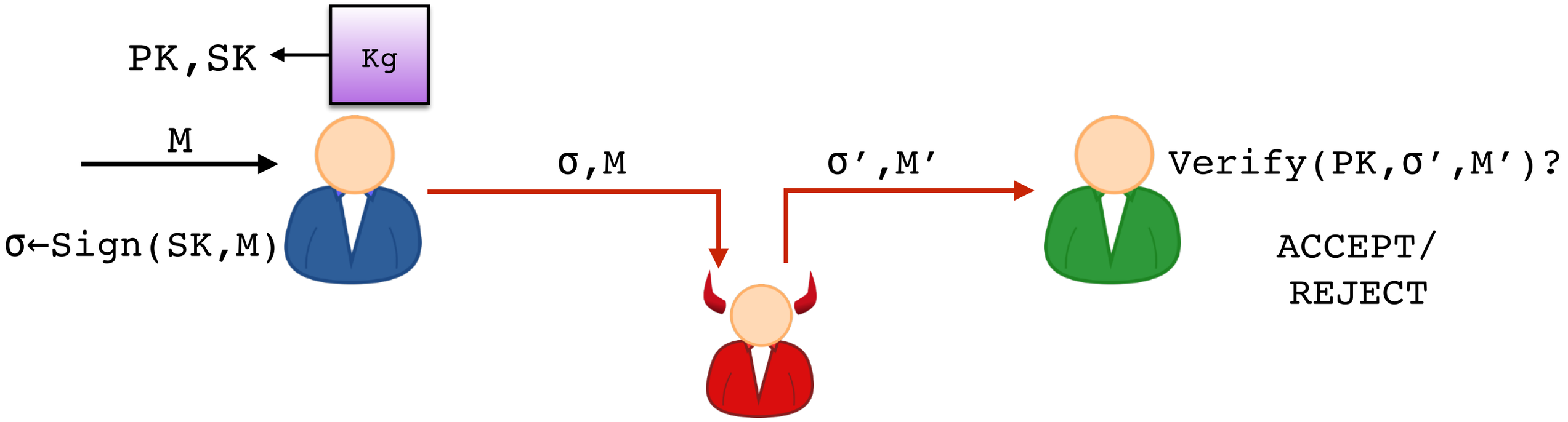
- Key generation algorithm **`Kg`**, takes no input and outputs a (random) public-verification-key/secret-signing key pair **`(PK,SK)`**

- Signing algorithm **`Sign`**, takes input the secret key **`SK`** and a message **`M`**, outputs "signature" **`σ←Sign(SK,M)`**

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

`Verify(PK,M,σ)=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


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

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

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

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`.

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

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

`M=1` weakness: If `M'=1` then it is easy to forge. Take `σ'=1`:

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

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 `σ'=2`.

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

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

Further "Plain" RSA Weaknesses


Broken


Assume `e=3`.

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

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

***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 mod N)`, and `σ'=(2*σ mod 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

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

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

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

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

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.`

RSA Signature Padding: PKCS #1 v1.5

Note: We already saw PKCS#1 v1.5 e*ncryption* 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 $\sigma = X^d \bmod N$

`Verify((N,3),M,σ)`:

1. $X \leftarrow (\sigma^3 \bmod N)$
2. Parse `X→aa||bb||Y||cc||digest`
3. If `aa≠00` or `bb≠01` or `cc≠00`
   or $Y \neq (FF)^{n-m-3}$
   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 e*ncryption* 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 $\sigma = X^d \bmod N$

Verify((N,3),M,σ):

1. X←($\sigma^3 \bmod N$)
2. Parse X→aa||bb||Y||cc||digest
3. If aa≠00 or bb≠01 or cc≠00
   or $Y \neq (FF)^{n-m-3}$
   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

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

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

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

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

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

$$(\sigma')^3 = 00\ 01\ FF\ 00\ H(M')\ \langle JUNK \rangle \bmod N$$

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:

$$00\ 01\ FF\ 00\ H(M')\ 00\ \ldots\ 00 \le (\sigma')^3 \le 00\ 01\ FF\ 00\ H(M')\ FF\ \ldots\ FF$$

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

Fun! (Assignment 2)

Steps in Attack

1. Pick $\mathbf{M}$ you want to forge a signature on.
2. Compute lower and upper bounds (numbers), using $\mathbf{H(M)}$.
3. Find a perfect cube $\mathbf{x}$ within allowed range.
4. Output cube root of $\mathbf{x}$ as forged signature $\boldsymbol{\sigma}$.

Why do so many people 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.

- Note: Attack (and others) defeated by using large `e=65537`
    - Example of defense-in-depth

Other RSA Padding Schemes: Full Domain Hash

`N`: `n`-byte long integer.
`H`: Hash fcn with `m`-byte output.
`k = ceil((n-1)/m)`

← Ex: `SHA-256, m=32`

`Sign((N,d),M)`:

1. $X \leftarrow 00||H(1||M)||H(2||M)||\ldots||H(k||M)$
2. Output $\sigma = X^d \bmod N$

`Verify((N,e),M,σ)`:

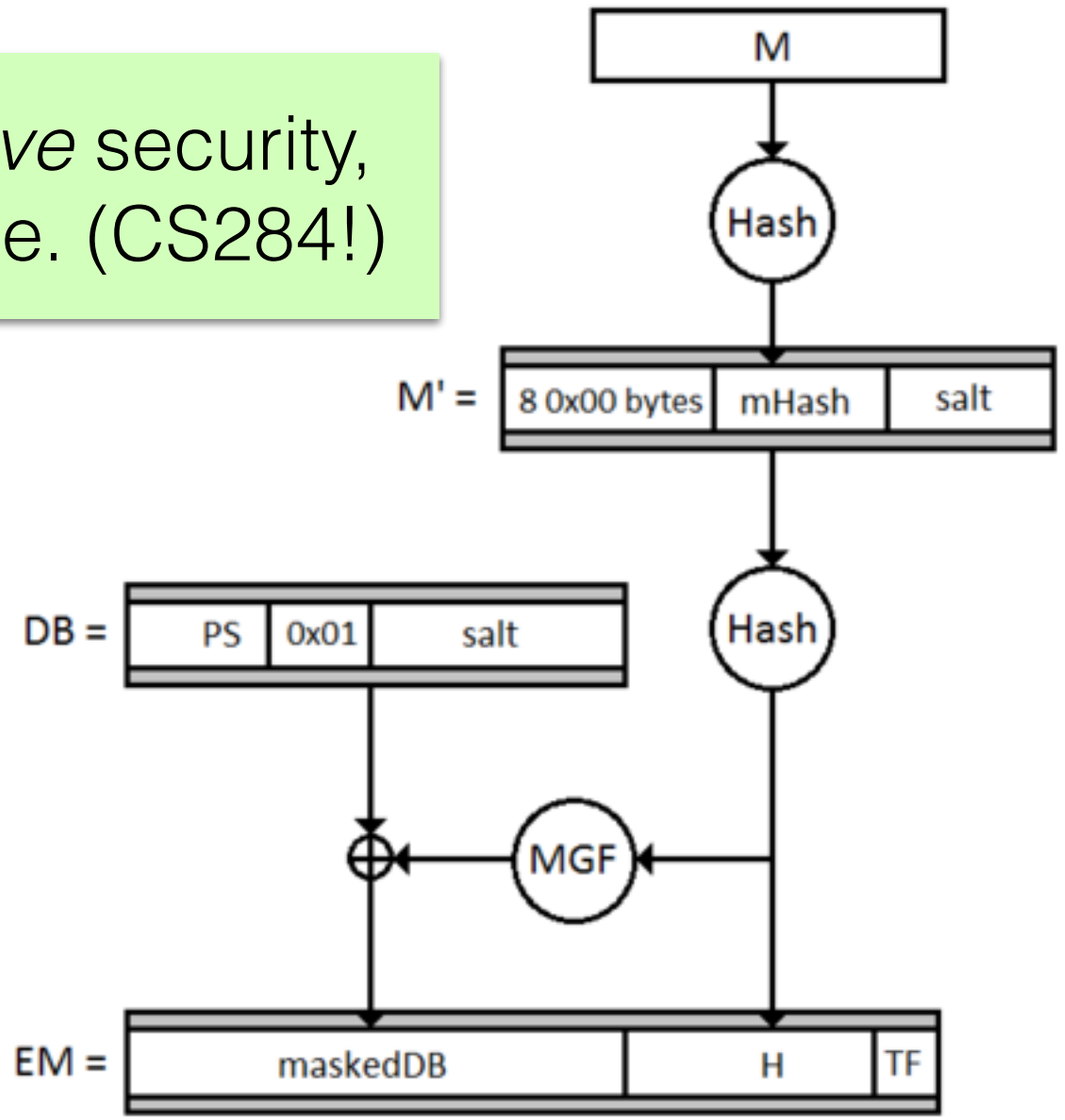
1. $X \leftarrow 00||H(1||M)||H(2||M)||\ldots||H(k||M)$
2. Check if $\sigma^e = X \bmod N$

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

Other RSA Padding Schemes: PSS (In TLS 1.3)

- Somewhat complicated
- *Randomized* signing

Bonus: 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

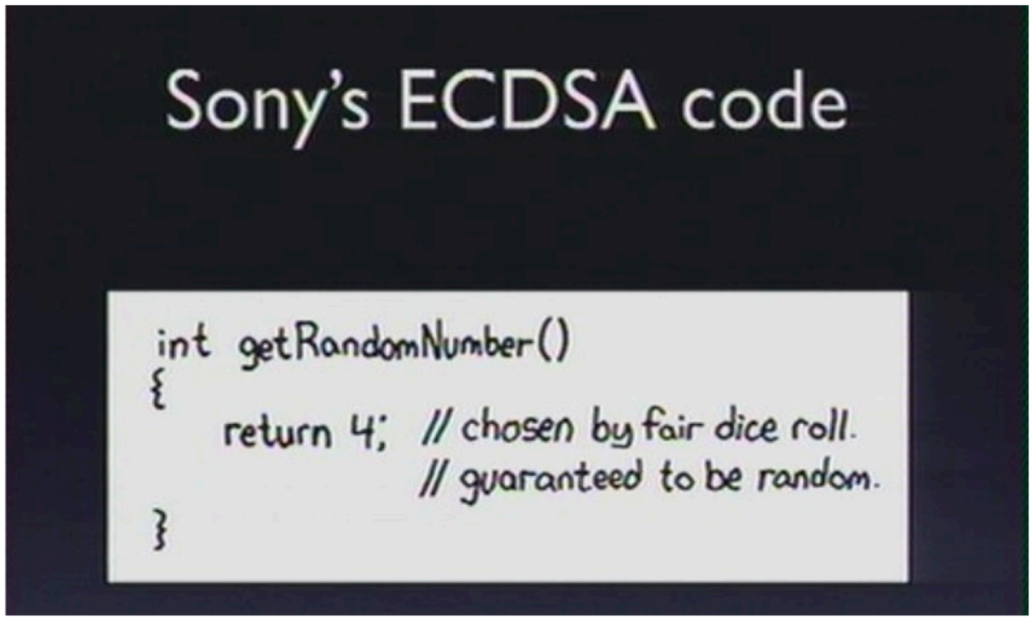
- Based on ideas related to Diffie-Hellman key exchange
- Secure, but even more ripe for implementation errors

—

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

Sean Hollister
12.29.10

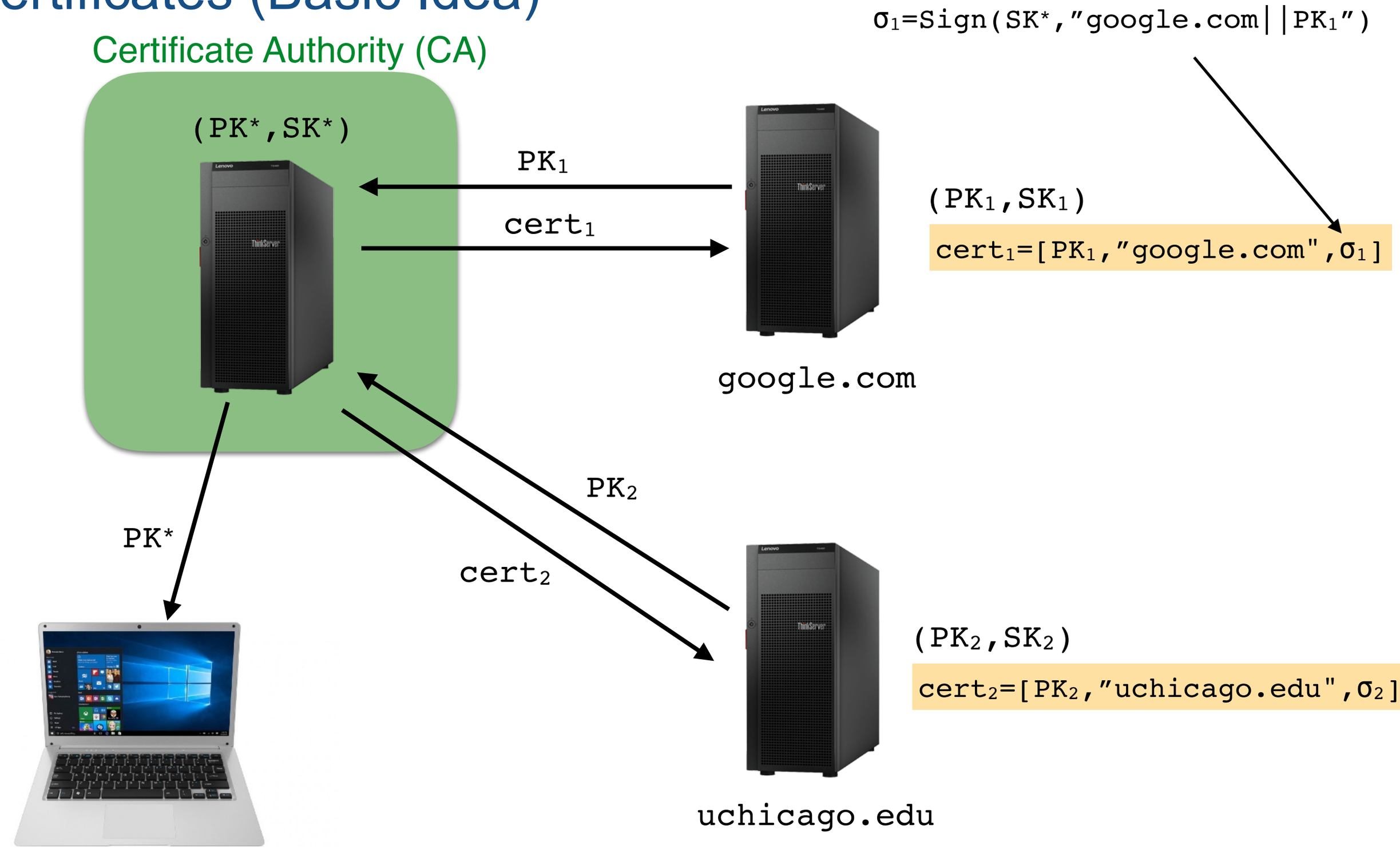
2
Shares



Public-Key Infrastructure (PKI)

- Main application for digital signatures are *certificates*, used in TLS and other protocols
- Used to support a “public-key infrastructure”

Certificates (Basic Idea)



- Trusted CA “issues certs”, i.e. signs public keys of other orgs.

Certificates (Basic Idea)

- Certificates in general are a tuple **`(PK,metadata,σ)`**
  - **`PK`** is public-key (may be for encryption, or for signature verification)
  - The **`metadata`** domain name, company info, sometimes addresses, crypto protocols to use, expiration date, etc.
  - **σ** is a signature on **`PK+metadata`** under CA's signing key.
- Issuing a cert involves varying levels of due diligence by CA
- If CA is negligent, then entire system is not trustworthy!

Authenticated Key Exchange with Certs

CA's verification key **PK***

(Pick random AES key **K**)

K

ClientHello

$cert_1$=[PK,"google.com",σ*]

Challenge: R

σ =Sign(PK,R)

C = Enc(PK_1,K)

PK,SK

Dec(SK,C)

K

PK* correct⇒ **PK_1** correct⇒ Person-in-the-Middle defeated!

Authenticated Key Exchange with Certs

CA's verification key **PK***

(Pick random AES key **K**)

K

ClientHello → **ClientHello** →

PK,SK

K←Dec(SK,C)

K

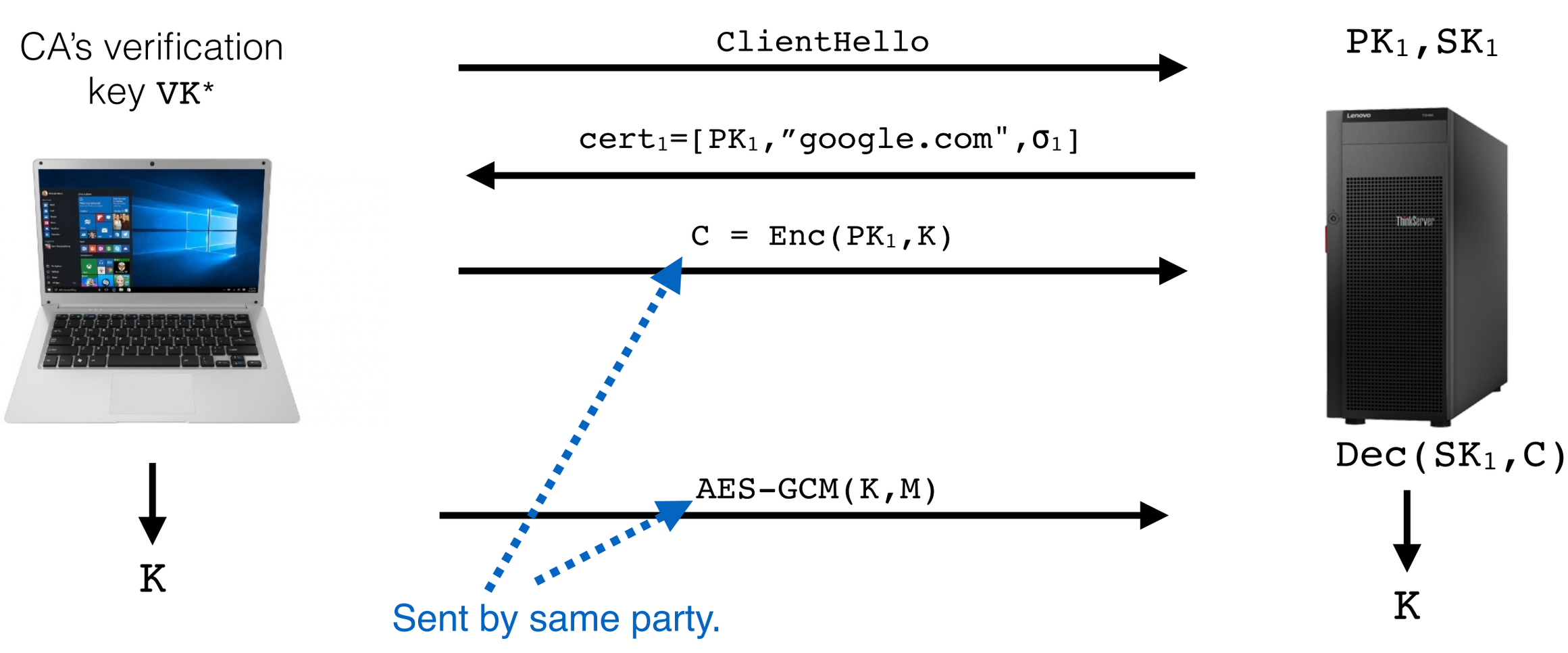
← cert= [**PK'**,"google.com",**?**]

← cert= [PK,"google.com",σ]

Adversary must forge signature, or trick CA into issuing cert.

Authenticated Key Exchange Notes

- Authentication is “unilateral” or “one-sided”
  - You are convinced you’re talking to **`google.com`**, but **`google.com`** has no idea who they are talking to.
  - However **`google.com`** knows they are continuing to talk to whoever sent **`C`**
  - You convince **`google.com`** of your identity using a password, not TLS.



The End