CS162: Operating Systems and Systems Programming

Lecture 28: Security

6 August 2015

Charles Reiss

https://cs162.eecs.berkeley.edu/

What is Computer Security?

Computing in the presence of an **adversary**!

Reliability, robustness, and fault tolerance

- Dealing with Mother Nature (*random* failures)

AND Security

- Dealing with actions of a ***knowledgeable*** attacker **dedicated** to causing harm

Malice, not just mischance

Protection vs. Security

Protection: mechanisms for controlling access of programs, processes, or users to resources

- Page table mechanism; round-robin schedule; data encryption

Security: **use of protection** to prevent **misuse** of resources

- Misuse defined with respect to ***policy***
  - E.g.: prevent exposure of certain sensitive information
  - E.g.: prevent unauthorized modification/deletion of data
- Need to consider external environment the system operates in
  - Will user accidentally reveal password?

An adversary?

Malicious – the adversary will always do the ***worst thing*** they can do

Key question – what access do they have?

- Can they replace my keyboard?
- Can they intercept all my network communications?
- Can they grab my hard disk when I'm not using it?

"Misuse of resources"?

Depends on the system!

Is it okay if other people can read my files?

- Web server: yes! (some of them)
- My laptop: no!

Is it okay if other people can store data on my machine?

- Mail server: yes!
- My laptop: no!

Example Security Tasks

Authentication

- Ensures that a user is who is claiming to be

Data integrity

- Ensure that data is not changed from source to destination or after being written on a storage device

Confidentiality

- Ensures that data is read only by authorized users

Non-repudiation

- Sender/client can’t later claim didn’t send/write data
- Receiver/server can’t claim didn’t receive/write data

Authorization: Who Can Do What?

How do we decide who is authorized to do actions in the system?

Authorization vs Authentication

How do we decide who is authorized to do actions in the system?

Assumes **we already know who the user is**!

Authorization: Who Can Do What?

How do we decide who is authorized to do actions in the system?

Access Control Matrix: contains all permissions in the system

- Resources across top
  - Files, Devices, etc…
- Domains in columns
  - A domain might be a user or a group of users
  - E.g. above: User D3 can read F2 or execute F3
- In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Authorization: Two Implementation Choices

Access Control Lists: store permissions with object

- Who (what processes) are allowed to do what to this object?

Capability List: each process tracks which objects has permission to touch

- What objects am I (my process) allowed to do things with?
- "Keys" to the object – sometimes can be passed between processes

Capability List

Example: page table

- List of *physical pages* process has access to

Example: file descriptors

- List of *open files* process has access to

Trick: List of objects that can be used can be same as "handle" to access objects

- Doesn't have to work this way

Capabilities as Addresses

One approach: give each object a long random ID

Its ID is a capability to access it

- Like file descriptor is capability to access file

ID can be passed around freely

- Perhaps even over a network

Can't be forged – can't guess random number

Access Control Lists

Store permissions with object

Example: NTFS files – in metadata for each file

- List of (user or group, read or write or execute or …)

Example: Unix files – in metadata for each file

- One user ("owner") – read and/or write and/or execute
  - Owner also has "change access control list" access
- One group – read and/or write and/or execute
- Default setting ("world") – read and/or write and/or execute

Access Control Lists

Cons:

- Maybe lots of users
- Settings scattered across many files

Authorization: Two Implementation Choices

Access Control Lists: store permissions with object

- Who **(what processes)** are allowed to do what to this object?

Capability List: each process tracks which objects has permission to touch

- What objects am I (my process) allowed to do things with?

Authorization: Combination Approach

Users have capabilities – **groups** or **roles**

Everyone with a particular role/group is "equivalent" for group resources

Each **process** is assigned one more users/groups/roles

Can access resources if any ACL entry matches

Objects have ACLs

- ACLs can refer to roles – users or groups
- Change object permissions object by modifying ACL
- Change broad user permissions via changes in group membership
- Possessors of proper credentials get access

Authorization: How to Revoke?

How does one revoke someone's access rights to a particular object?

- Easy with ACLs: just remove entry from the list
- Takes effect immediately

Harder to do with capabilities – how to find capability to change?:

- Not so bad in a single machine: could keep all capability lists in a well-known place (e.g., the OS capability table).
- Very hard in distributed system, where remote hosts may have crashed or may not cooperate (more in a future lecture)

Revoking Capabilities

Various approaches to revoking capabilities:

- Put expiration dates on capabilities and force reacquisition
- Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt)
- Maintain back pointers to all capabilities that have been handed out
- Maintain a revocation list that gets checked on every access attempt

Authorization on Unix

Combination approach:

- Two "capabilities" associated with process
- Checked against ACLs/static policies

Information in **process control block**:

- User ID
- Group ID list

Inherited on **fork**

Checked against file permissions stored in inode on open():

- User ID + read/write/execute
- Group ID + read/write/execute

Authorization on Unix: Files

User ID + Group ID in PCB

Checked against file permissions stored in inode on open():

- User ID + read/write/execute
- Group ID + read/write/execute

Limited access control list

Authorization on Unix: UID = UID

sigaction() + kill()?

- Rule: can only send signals caller's UID matches your UID

Changing file permissions?

- UID needs to match file "owner"

Very limited access control list

Authorization on Unix: Enforced where?

Checked on system call entry

Can't be checked in the system library, programs could cheat and substitute their own system library

Authorization on Unix: Superuser

User ID 0 is special:

- AKA *superuser*, *root*
- Some system calls only work as root – shutdown, mount new filesystems, etc.
- Automatically passes all permission checks

How does *login* work?

```
somemachine login: joeuser
Password: ********
```

How does this program work? It needs to...

Change user IDs, and

Check if the password is correct

How does *login* work?

```
somemachine login: joeuser
Password: ********
```

How does this program work? It needs to...

Change user IDs, and

Check if the password is correct

Changing user IDs

System call (with OS library wrapper):

```
int setuid(uid_t uid);
```

Changes UID of current process

Only works for superuser (mostly)

How does *login* work?

```
somemachine login: joeuser
Password: ********
```

How does this program work? It needs to...

Change user IDs, and

Read passwords

Local Unix password storage

`/etc/shadow`

- not readable anyone but superuser
- contains passwords processed with one-way function

Why a one-way function?

- Make it less bad if someone gets a copy of /etc/shadow
- They can still try *a lot* of passwords – trillions easily!

How does *sudo* work?

```
joeuser@somemachine$ sudo restart
Password: ********
```

sudo: utility to run a command with superuser permissions

- Started by non-superuser

Recall: **current UID inherited across fork()**

How does *sudo* work?

Extra file metadata on **executables**:

- set-user-ID

exec() system call changes user ID if this is set to the **owner of the executable**

sudo program is owned by root and has set-user-ID bit set

- Allows it to read /etc/shadow, config files

set-user-ID programs

"Gate" to higher privilege

- Give *controlled* access to extra functionality
- Make authorization policy *outside of the kernel*
- Make authentication policy (decide who is who) *outside of kernel*

Need to write such programs ***really*** carefully

- What if stdin, stdout, stderr start closed?
- What if PATH environment variable is set to something ridiculous?
- ...

Recall: Handling system calls safely

Same work as interrupts to enter/leave

Copy arguments from user registers/*memory (!)*

- **Carefully check memory locations!**

Validate arguments

- Kernel can't crash if user code is broken/malicious

Do operation

Copy result back into user registers/memory

- **Carefully check memory locations!**

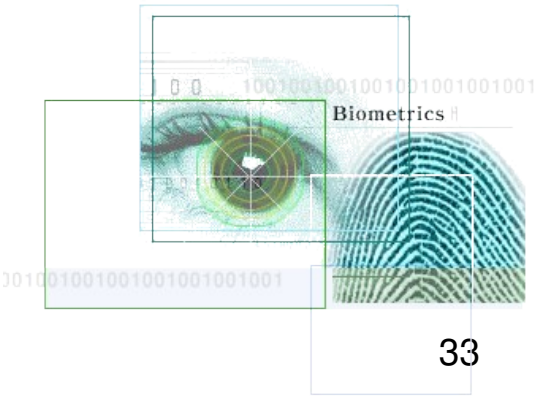
Authentication: Identifying Users

Passwords/passphrases

- Shared secret between two parties
- Typically typed by user

Unix model: Validated by *login*

- Usually stored in file only accessible to root
- Alternative approaches: network service (example: instructional labs)



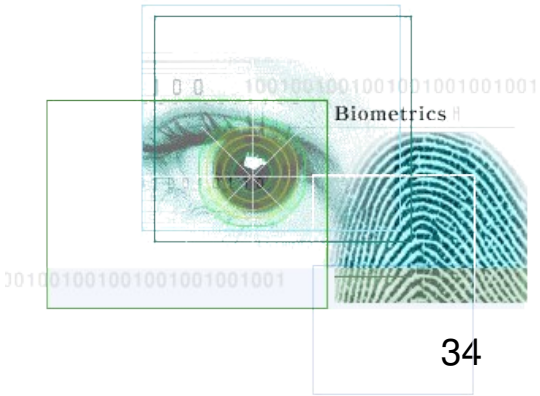
Authentication: Identifying Users

Authentication Tokens / Smart Cards

- Electronics embedded in card capable of providing long passwords or satisfying challenge/response queries
- Sometimes with display to show temporary password
- Typical goals: **can't be copied**; need to **have it physically to use it**

Biometrics

- Use of one or more intrinsic physical or behavioral traits to identify someone
- Examples: fingerprint reader, palm reader, retinal scan
- Can be copied
- Most useful if someone **will arrest you** if you're putting a piece of gel on the fingerprint scanner



Alternative Permission Models

Users and groups are very coarse-grained

Example: user on consumer machine probably can access everything that matters

Maybe better to have finer-grained permissions

- "Can access files in directory /home/user/browser-cache, the display, the keyboard, the audio device, but nothing else"

Some partial support for this, e.g. "sandboxing" in web browsers

Securing Communication: Cryptography

Cryptography: communication in the **presence of adversaries**

Studied for thousands of years

- See the Simon Singh's The Code Book for an excellent, highly readable history

Central goal: confidentiality

- How to encode information so that an adversary can't extract it, but a friend can

General premise: there is a key, possession of which allows decoding, but without which decoding is infeasible

- Thus, key must be kept secret and not guessable

An Untrusted Network

Assume **adversary *Eve*** on network can

- drop messages
- change messages
- read message
- repeat messages
- create messages with arbitrary content

But can't:

- access our machines except by sending messages on the network

Some Untrusted Storgae

Assume **adversary *Eve*** on filesystem can

- remove files
- change files
- read files
- copy files
- create files with arbitrary content

But can't:

- access our programs except by sending messages on the network (they're stored on a different filesystem)

Goal: The Secure Channel

Alice sends a sequence of messages to Bob

Goals:

- **Confidentiality**: Eve does not learn the contents of the messages
- **Authenticity**: Bob receives messages only from Alice with the contents she choose
- Bob knows what order the messages come in and if there are holes

Idea: Like a private wire from Alice to Bob

Do we have enough?

If Eve and Alice start with the same information...

How can Bob tell who sent him a message?

Answer: He can't.

One solution: Shared secrets

Bob needs to know *something* about Alice securely (or vice-versa).

One example:

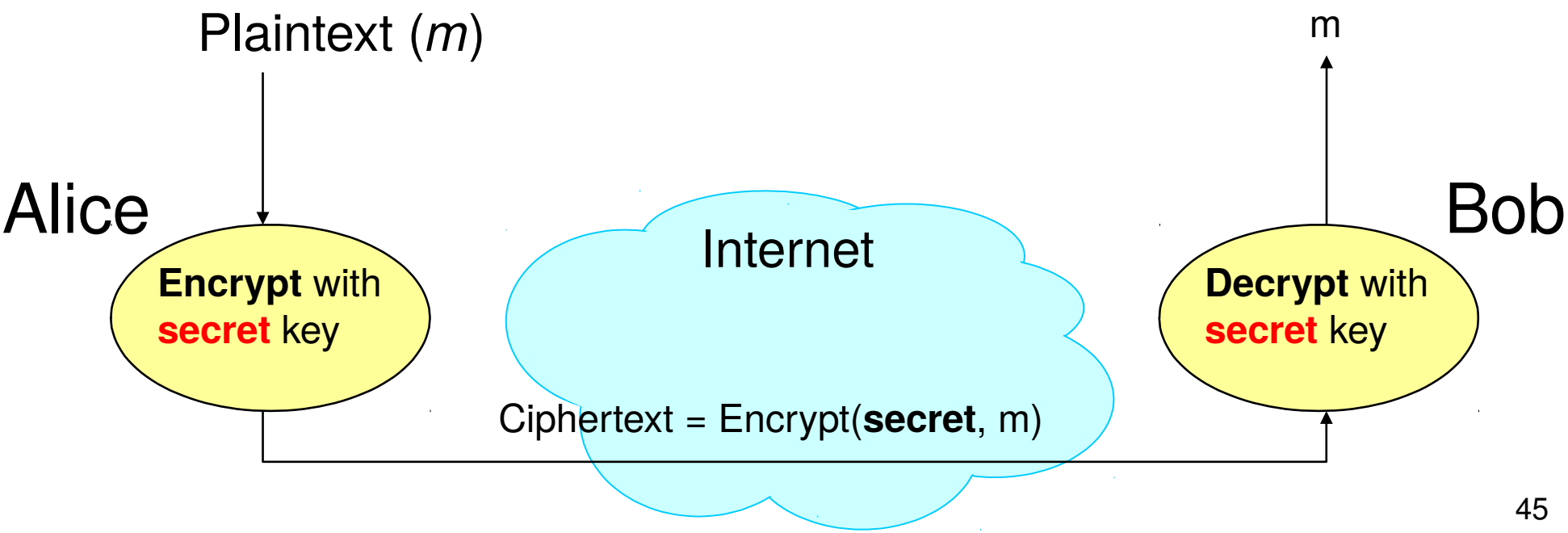
- They agree on a password in advance (knowing Eve isn't watching)

Symmetric Key Encryption

Math to rescue!

- *encryption function* E(*key*, *message*)
- *decryption function* D(*key*, *message)*

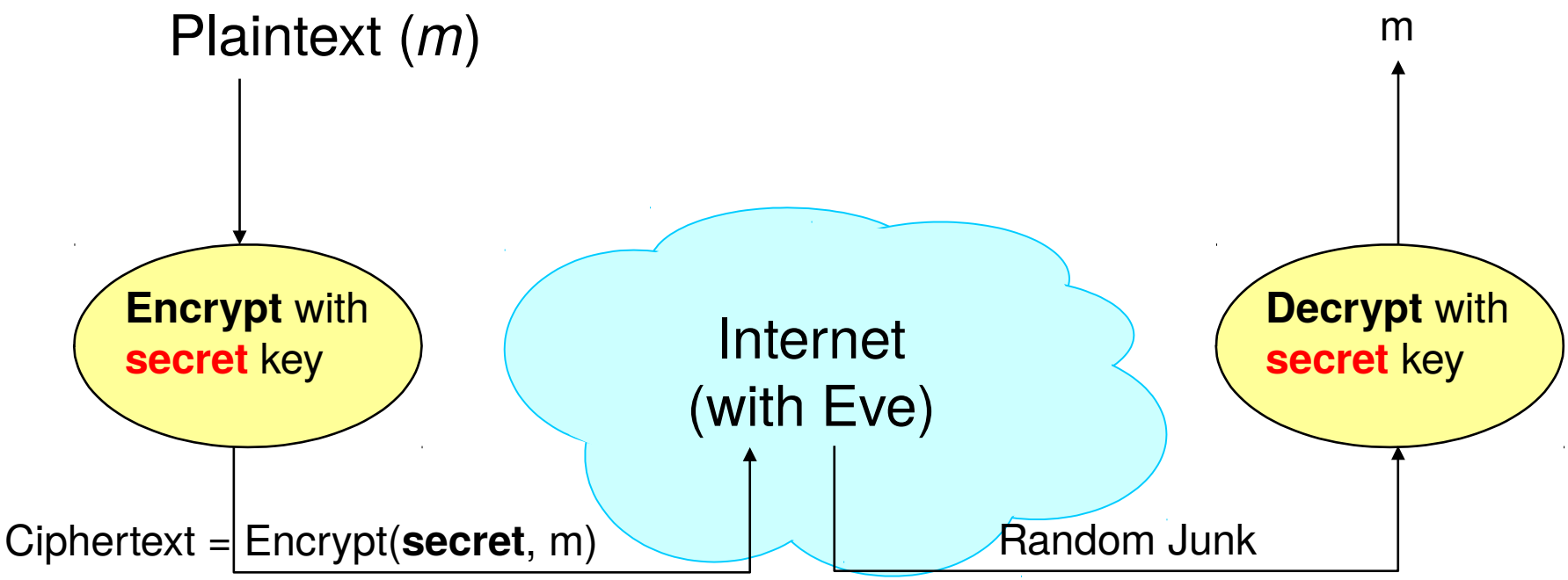
Confidentality: Eve can't learn anything about *message* or *key* from E(key, message) without knowing key



Encryption != authenticatoin

Problem: What if Eve sends junk? Is that a valid message?

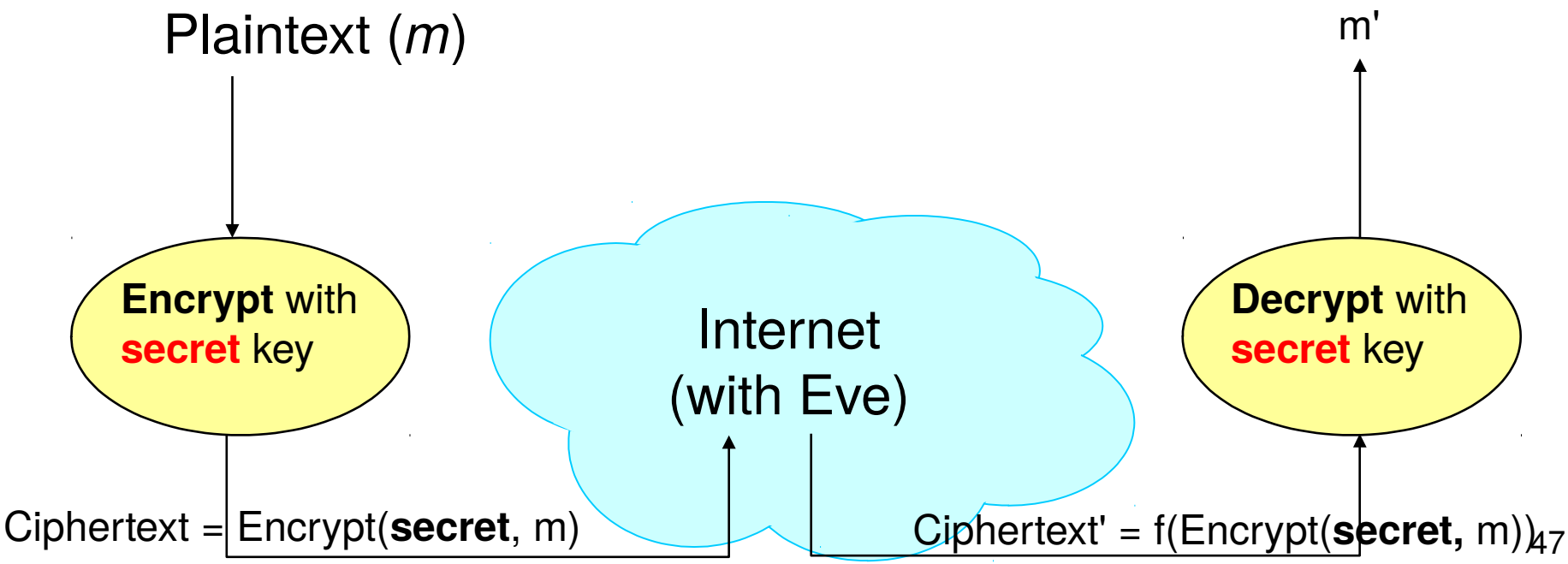
What will Decrypt(**secret**, <u>random junk</u>) be?



Encryption != authenticatoin

Eve might be able to manipulate messages ***meaningfully***

- "Transfer $10 to Eve"
- Example: Change value to something else by replacing that part of the message with junk (or can do even better, sometimes!)

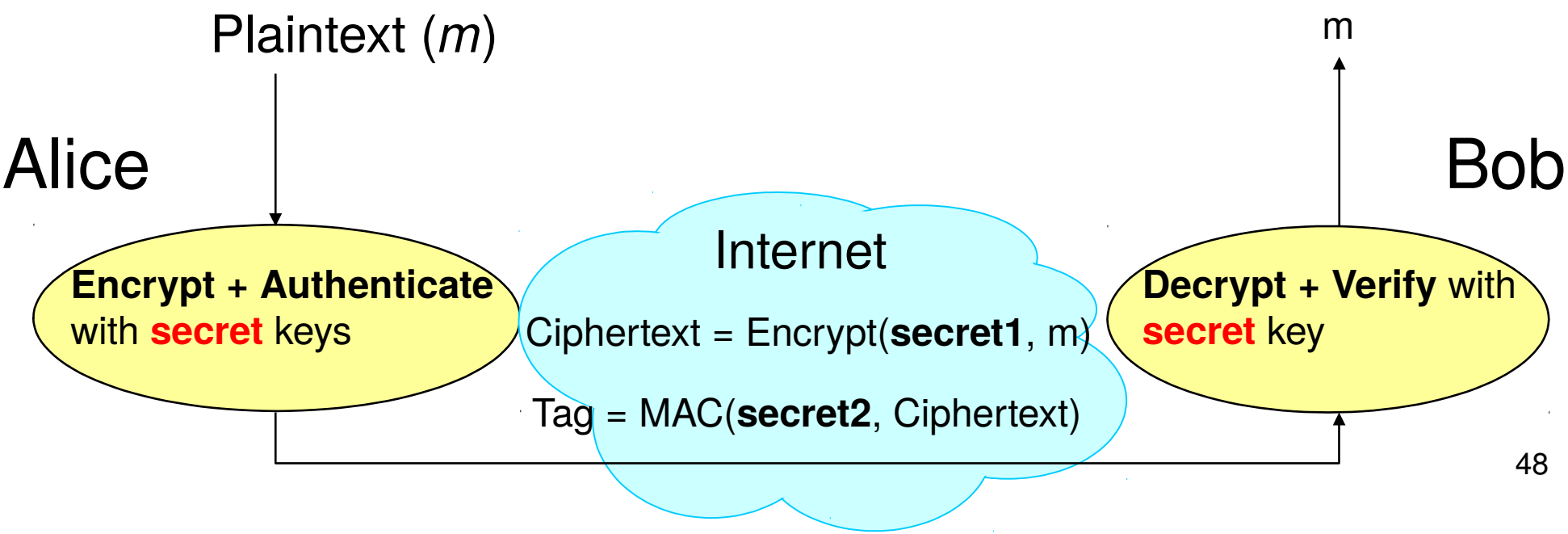


Symmetric Key Authentication: **M**essage **A**uthentication **C**odes

More math to the rescue!

- *authenticate:* tag = **MAC**(*key*, *message*)

Authentication: Eve can't produce the correct *tag* without knowing the correct *tag*

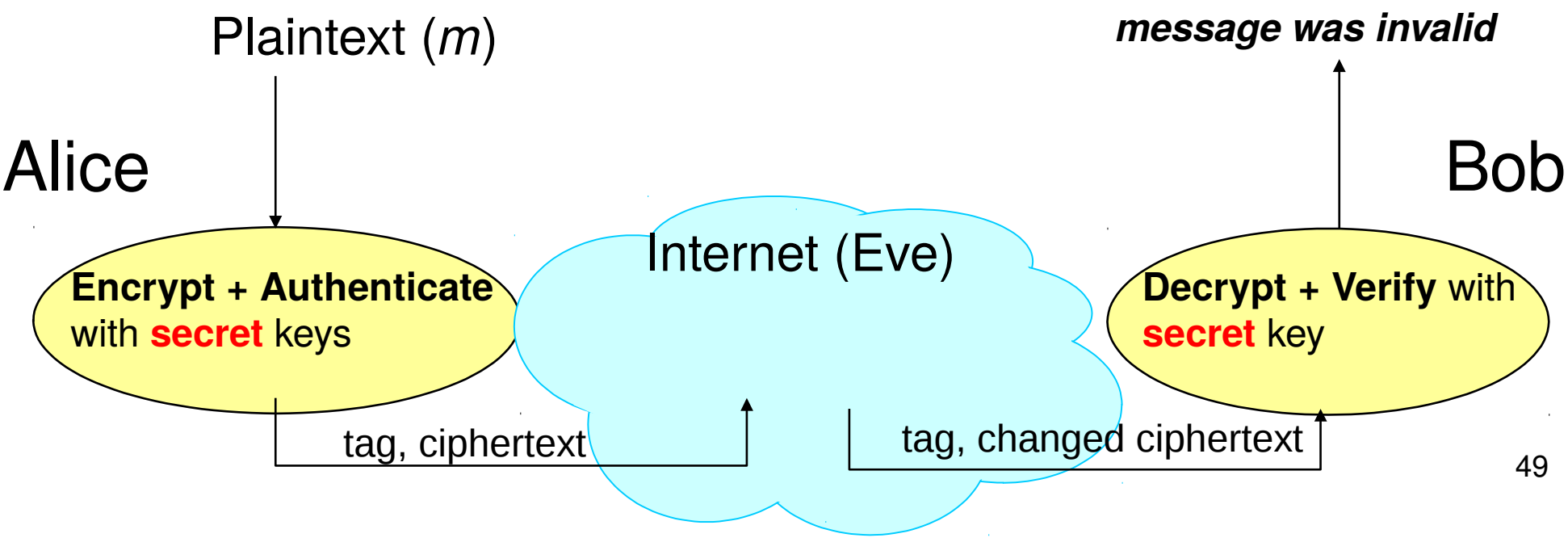


Symmetric Key Authentication: **M**essage **A**uthentication **C**odes

More math to the rescue!

- *authenticate:* tag = **MAC**(*key*, *message*)

Authentication: Eve can't produce the correct *tag* without knowing the correct *tag*



Authenticated Encryption

ciphertext, tag = EncryptThenMAC(key, message)

invalid or message =
ValidateThenDecrypt(key, message)

Combine and encryption and MAC function

- AES-GCM = *encryption* AES-CTR + *MAC* AES-GMAC
- AES-CBC + HMAC-SHA256
- (Caution: want independent keys!)

Authenticated Encryption

Cryptographic **primitives**: Building blocks!

Example: **AES** (Advanced Encryption Standard):

AES-Encrypt(key, 128-bit block X) = 128-bit block Y

AES-Decrypt(key, 128-bit block Y) = 128-bit block X

Use these to build encryption, authentication for arbitrary-length messages

- *AES*-GCM = *encryption* *AES*-CTR + *MAC* *AES*-GMAC
- *AES*-CBC + HMAC-*SHA256*
- (Caution: want independent keys!)

Recall: The Secure Channel

Alice sends a sequence of messages to Bob

Goals:

- **Confidentiality**: Eve does not learn the contents of the messages
- **Authenticity**: Bob receives messages only from Alice with the contents she choose
- ***Bob knows what order the messages come in and if there are holes***

Idea: Like a private wire from Alice to Bob

Guaranteeing Order

Bob knows what order the messages come in and if there are holes

One solution: All messages start with a message number

- Bob needs to remember the last message number used with the secret key
- Note that Eve can resend a very old message, so Bob can't reset the numbering without choosing a new key

Public Key Cryptography

Problem: Need to agree on a secret

Can we avoid that?

Alternative: ***key pair***

- One key Bob can publish to everyone (Alice and Carol) – a **public key**
- One key Bob can keep to himself *only* – a **private key**

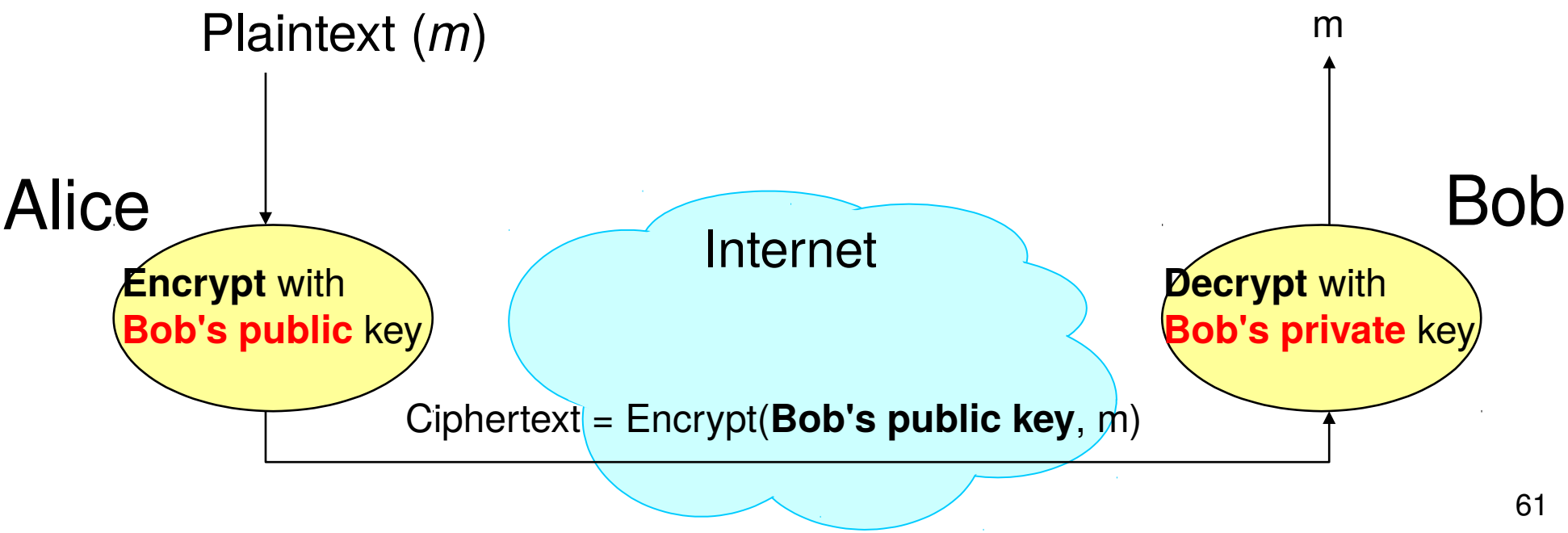
Advantage:

- Can distribute key to *everybody*

Asymmetric Encryption

Encrypt(public-key, message) = ciphertext

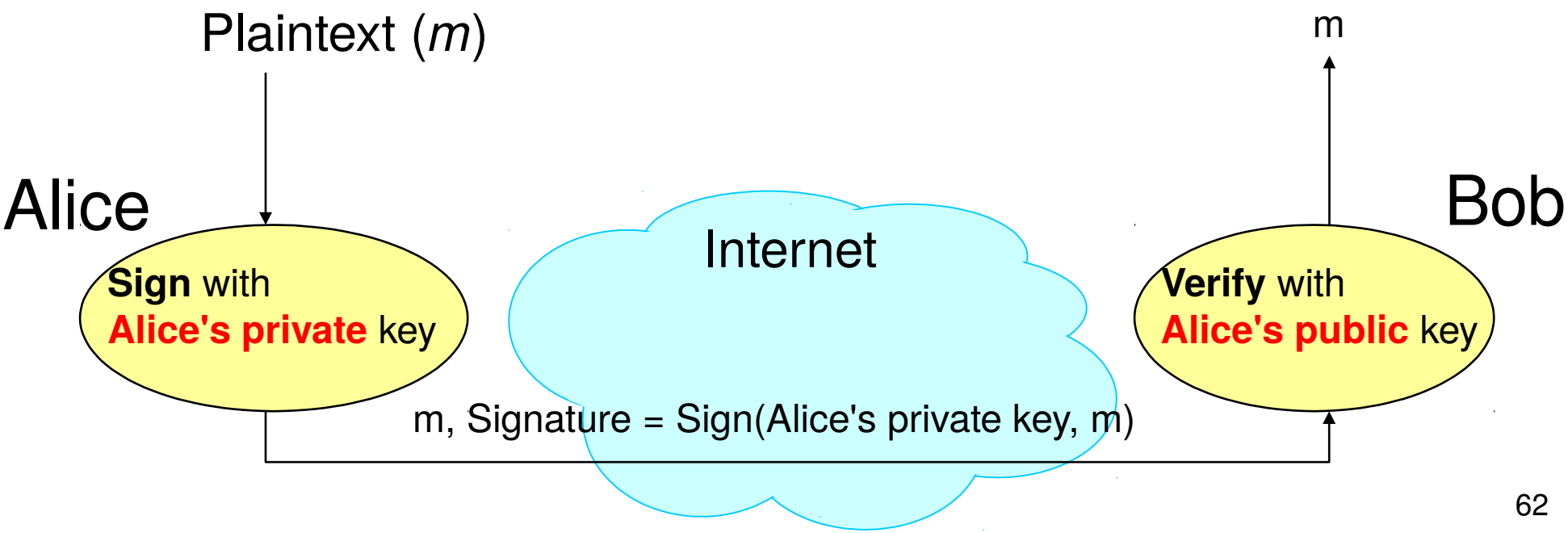
Decrypt(private-key, ciphertext) = message



Asymmetric Authentication AKA Digital Signatures

Sign(**private**-key, message) = *signature*

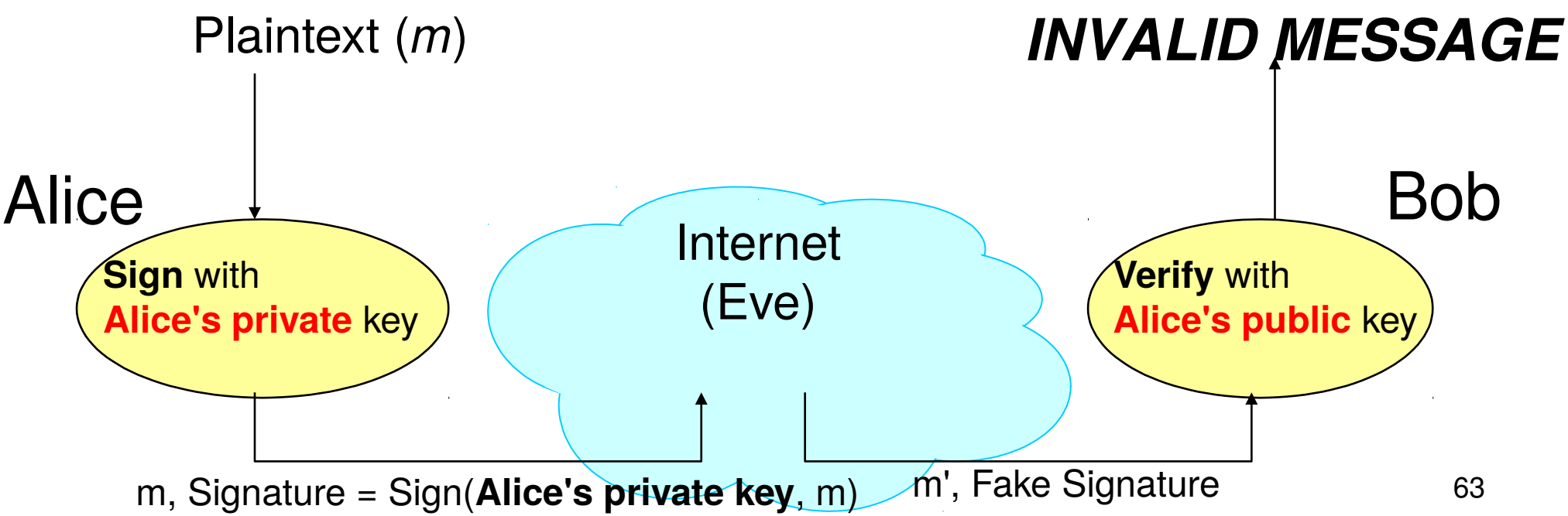
Verify(public-key, signature, message) = yes or no



Asymmetric Authentication AKA Digital Signatures

Sign(**private-key**, message) = *signature*

Verify(public-key, signature, message) = yes or no



Bootstraping Symmetric Crypto

Alice wants to share a *secret* with Bob?

- Encrypt(Alice's public key,
  "`The password is` **`bar`**")

Then use symmetric encryption + authentication

Bootstraping Asymmetric Crypto

Suppose: Alice and Bob both trust Trent and know his public key.

Trent produces this message:

- Sign(Trent's private key,
  "Alice's public key is XXXXXXXX")
- Called a ***certificate***

Then Bob can send this to Alice.

Trent is called a ***certificate authority***

Certificate Authorities in the Wild

List of Trents distributed with operating systems, browsers

Trusted to produce public keys for websites

Are they worthy of this trust????

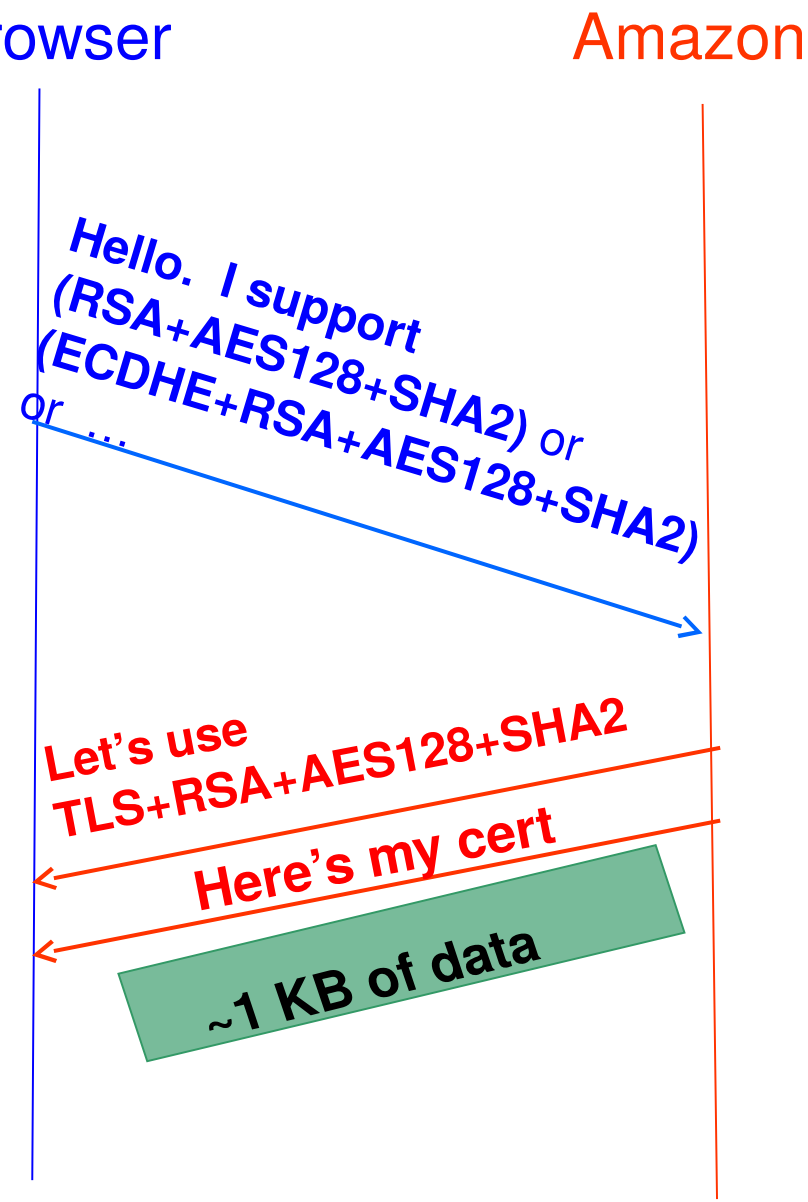
TLS / HTTPS (1)

HTTPS → "secure" HTTP

Handshake:

- Browser says what types of symmetric, asymmetric encryption it supports
- Server chooses from browser's list, sends **certificate**

Next step depends on what's selected



TLS / HTTPS (2)

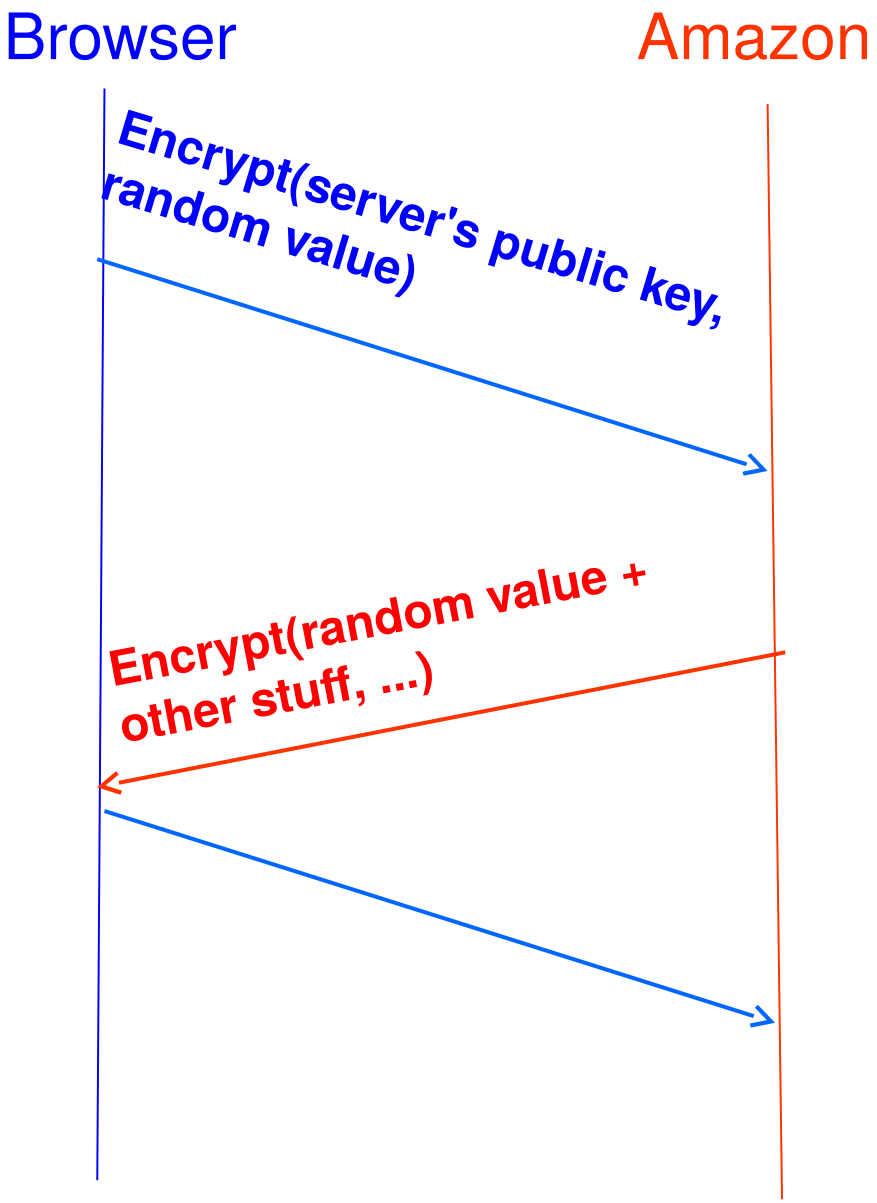
One example: RSA key exchange

Client sends a random value encrypted to the key *in the certificate*

This value is used to create **symmetric keys**

Not the best approach in TLS: want "forward secrecy"

- Use temporary key pairs so compromising the permanent key won't allow decryption after the fact



Public Key Cryptography (1)

Invented in the 1970s

- Revolutionized cryptography
- (Was actually invented earlier by British intelligence)

How can we construct an encryption/decryption algorithm using a key pair with the public/private properties?

- Answer: Number Theory

Generally **slower, larger messages than symmetric crypto**

Public Key Crypto Algorithms

Based on modular exponentiation of large integers:

- RSA-based
  - Rivest / Shamir / Adleman, 1977; RFC 3447
  - Probably most widely used
- DH-based schemes (ElGamal, DSA, ...)

Elliptic curve-based approaches:

- Mathematically similar to DH-based schemes
- Based on "curves" (fields in the abstract Algebra sense) in a Galois-field space
- Shorter keys and signatures and encrypted messages than modular-exponentiation-based approaches

(and more, less common ones)

Logistics

Project 3 Due Monday

HW3 due next Wednesday

Final exam next Thursday

Final review Tuesday and discussion M/Tu

Break