



University of
Pittsburgh

Applied Cryptography and Network Security

CS 1653



Summer 2023
Sherif Khattab
ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

Announcements

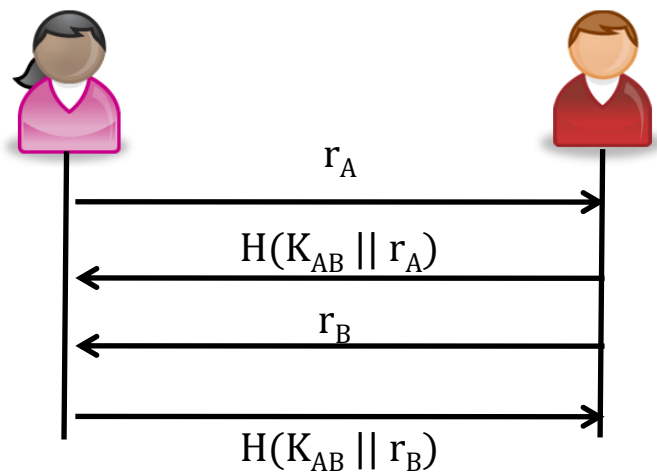
- Homework 3 due this Friday @ 11:59 pm
- Phase 2 of Project
 - posted tonight
 - due on Tuesday 6/27 @ 11:59 pm
- Next week
 - no lecture on Monday
 - Makeup lecture on Friday 6/16 @ 11:00 am

What can we do with a cryptographic hash function?

Document Fingerprinting

Use $H(D)$ to see if D has been modified

Example: GitHub commit hashes



Mutual Authentication

Message Authentication Code (MAC)

- Assume a shared key K
- Sender:
 - Compute $c = E_K(H(m))$
 - Transmit m and c
- Receiver:
 - Compute $d = E_K(H(m))$
 - Compare c and d

Hash functions can even be used to generate cipher keystreams

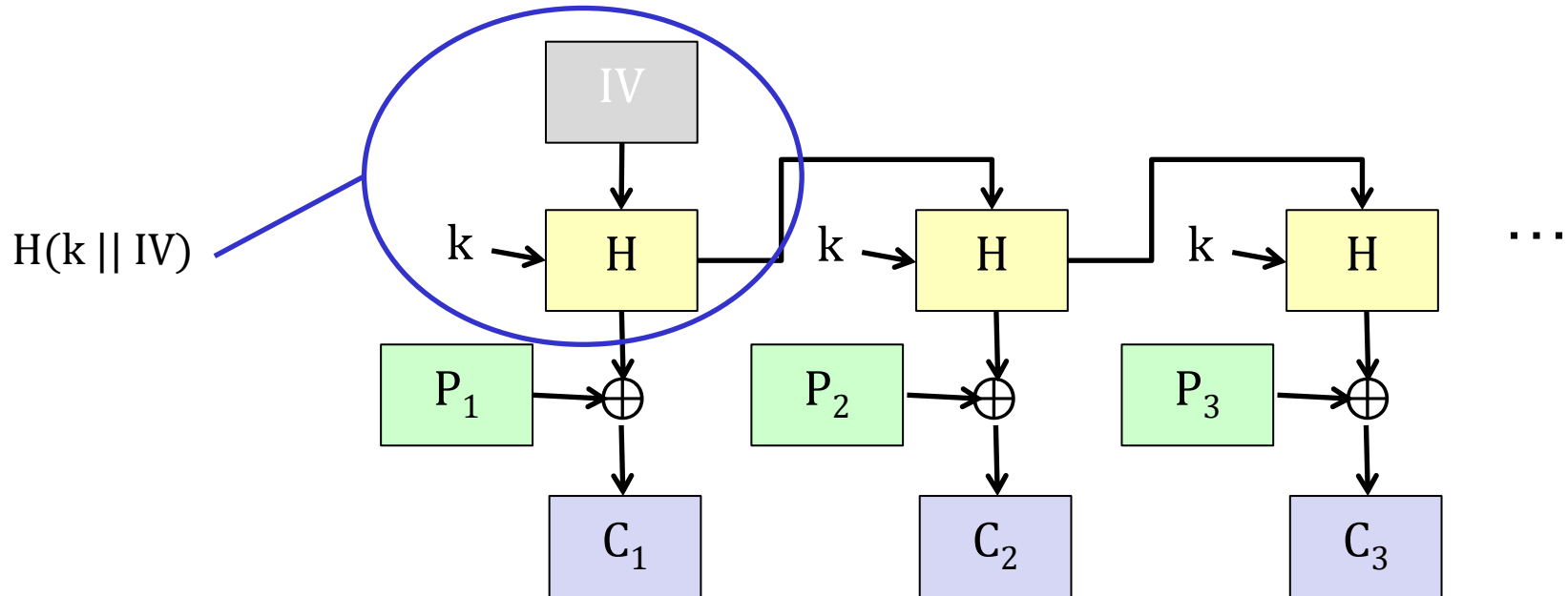
Question: What block cipher mode does this remind you of?

Output feedback mode (OFB)

Why is this safe to do?

Remember that hash functions need to behave “randomly” in order to be used in cryptographic applications

Even if the adversary knows the IV, they cannot figure out the keystream without also knowing the key, k



Hash functions also provide a means of safely storing user passwords

Consider the problem of safely logging into a computer system

Option 1: Store $\langle \text{username}, \text{password} \rangle$ pairs on disk

Correctness: This approach will certainly work

Safety: What if an adversary compromises the machine?

All passwords are leaked!

This probably means the adversary can log into your email, bank, etc...

Option 2: Store $\langle \text{username}, H(\text{password}) \rangle$ pairs on disk

Correctness:

Host computes $H(\text{password})$

Checks to see if it is a match for the copy stored on disk

Safety: Stealing the password file is less* of an issue

The importance of hash function's cryptographic properties

1. **Preimage resistance:** Given a hash output value z , it should be infeasible to calculate a message x such that $H(x) = z$

Without this, we could recover hashed passwords!



2. **Second preimage resistance:** Given a message x , it is infeasible to calculate a second message y such that $H(x) = H(y)$

Example: File integrity checking

Say the `ls` program has a fingerprint f

We could create a malicious version of `ls` that actually executes `rm -rf *`, but has the same document fingerprint

3. **Collision resistance:** It is infeasible to find two messages x and y such that $H(x) = H(y)$

Later on, we'll see that this can lead to attacks that let us inject arbitrary content into protected documents!



How do hash functions actually work?

It is perhaps unsurprising that hash functions are effectively compression functions that are iterated many times

- **Compression:** Implied by the ability to map a large input to a small output
- **Iteration:** Helps “spread around” input perturbations

The KPS book spends a lot of time talking about the “MD” family of message digest functions developed by Professor Ron Rivest (MIT)

Bad news: the most recent MD function, MD5, was broken in 2008

- Specifically, it has been shown possible to generate MD5 collisions in $O(2^{32})$ time, which is **much** faster than the theoretical “best case” of $O(2^{64})$
- We’ll talk more about this later in the course (~Week 9)

We’ll focus on SHA-1 (officially deprecated by NIST in 2011)

SHA-1 is built using the Merkle-Damgård construction

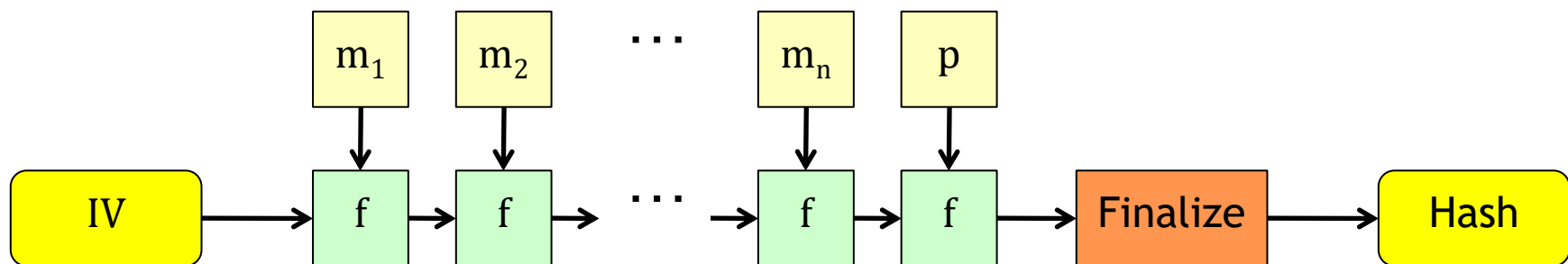
The **Merkle-Damgård construction** is a “template” for constructing cryptographic hash functions

- Proposed in the late ‘70s
- Named after Ralph Merkle and Ivan Damgård

Essentially, a Merkle-Damgård hash function does the following:

1. Pad the input message if necessary
2. Initialize the function with a (static) IV
3. Iterate over the message blocks, applying a **compression function** f
4. Finalize the hash block and output

Why is a static IV ok?



Merkle and Damgård independently showed that the resulting hash function is **secure** if the compression function is **collision resistant**

SHA-1: a thousand-mile view...

Input: A message of bit length $\leq 2^{64} - 1$ (will see why soon)

Output: A 160-bit digest

Steps:

- Pad message to a multiple of 512 bits

- Initialize five 32-bit words of state

 - A, B, C, D, E in the diagram (**5x32 = 160 bits**)

- For each 512-bit chunk of input message

 - Expand the sixteen 32-bit words into 80 32-bit words

 - Apply function at right eighty times to change state

- Concatenate the state to produce output

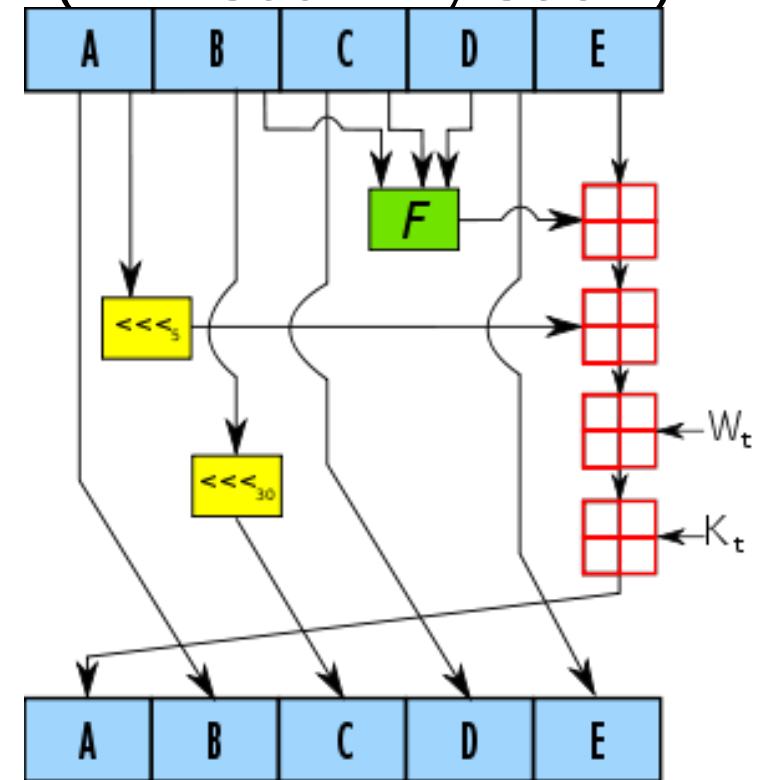


Image from Wikipedia

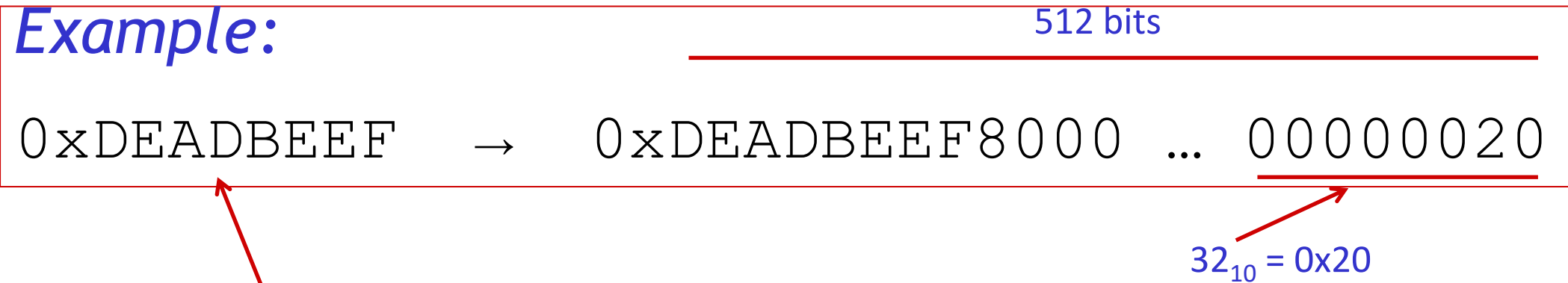
SHA-1: Initialization and Padding

Pre-processing:

- append the bit '1' to the message
- append $0 \leq k < 512$ '0' bits, so that the resulting message length (in bits) is congruent to $448 \equiv -64 \pmod{512}$
- append length of message (before pre-processing), in bits, as 64-bit big-endian integer

SHA-1: Initialization and Padding

Example:



Initializing the compression function

Process the message in successive 512-bit chunks:

break message into 512-bit chunks

for each chunk

break chunk into sixteen 32-bit big-endian words $w[i]$, $0 \leq i \leq 15$

Extend the sixteen 32-bit words into eighty 32-bit words:

for i from 16 to 79

$w[i] = (w[i-3] \text{ xor } w[i-8] \text{ xor } w[i-14] \text{ xor } w[i-16]) \lll 1$

Initialize hash value for this chunk:

$a = h0 = 0x67452301$

$b = h1 = 0xEFCDAB89$


$c = h2 = 0x98BADCFE$

$d = h3 = 0x10325476$

$e = h4 = 0xC3D2E1F0$

Note: \lll denotes a left rotate.

Example: $00011000 \lll 4$


 10000001

Note: These variables comprise the internal state of SHA-1. They are continuously updated by the compression function, and are used to construct the final 160-bit hash value.

Main body of the compression function

Main loop:

```
for i from 0 to 79
```

```
if 0 ≤ i ≤ 19 then
```

```
f = (b and c) or ((not b) and d); k = 0x5A827999
```

```
else if 20 ≤ i ≤ 39
```

```
f = b xor c xor d; k = 0x6ED9EBA1
```

```
else if 40 ≤ i ≤ 59
```

```
f = (b and c) or (b and d) or (c and d); k = 0x8F1BBCDC
```

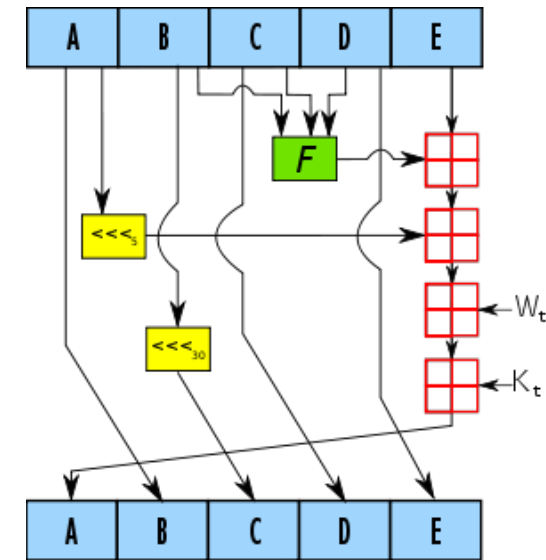
```
else if 60 ≤ i ≤ 79
```

```
f = b xor c xor d; k = 0xCA62C1D6
```

... but other times, it is treated as an unsigned integer

```
temp = (a <<< 5) + f + e + k + w[i]
```

```
e = d; d = c; c = b <<< 30; b = a; a = temp
```



Add this chunk's hash to result so far:

$$h_0 = h_0 + a; \quad h_1 = h_1 + b; \quad h_2 = h_2 + c; \quad h_3 = h_3 + d; \quad h_4 = h_4 + e$$

Finalizing the result

Produce the final hash value (big-endian):

output = h0 || h1 || h2 || h3 || h4



"||" denotes concatenation

Interesting note:

$$k_1 = 0x5A827999 = 2^{30} \times \sqrt{2}$$

$$k_2 = 0x6ED9EBA1 = 2^{30} \times \sqrt{3}$$

$$k_3 = 0x8F1BBCDC = 2^{30} \times \sqrt{5}$$

$$k_4 = 0xCA62C1D6 = 2^{30} \times \sqrt{10}$$

Question: Why might it make sense to choose the k values for SHA-1 in this manner?

SHA-1 in Practice

SHA-1 has fairly good randomness properties

- SHA1("The quick brown fox jumps over the lazy dog")
□ 2fd4e1c6 7a2d28fc ed849ee1 bb76e739 1b93eb12
- SHA1("The quick brown fox jumps over the lazy **cog**")
□ de9f2c7f d25e1b3a fad3e85a 0bd17d9b 100db4b3

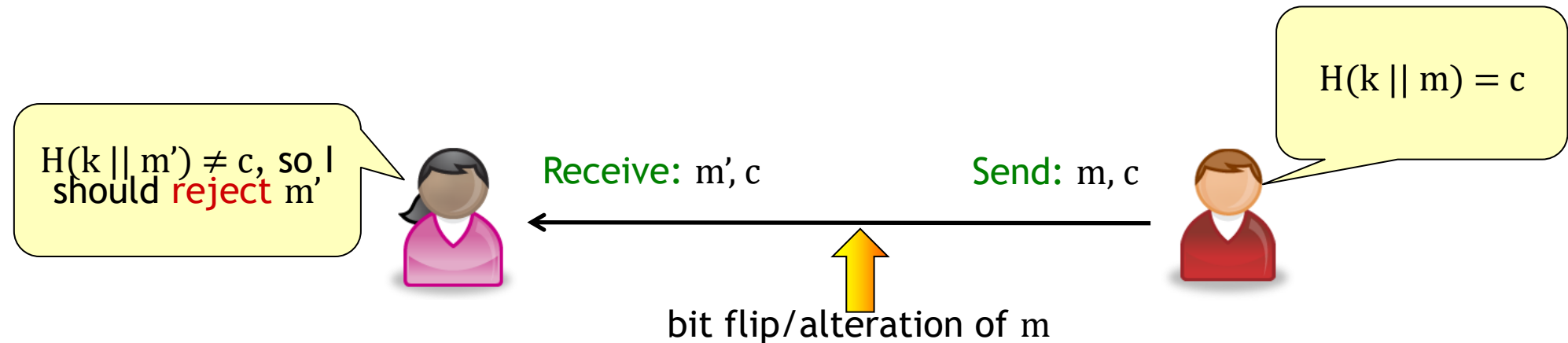
In the above example, changing 1 character of input alters 81 of the 160 bits in the output!

To date, the best attack on SHA-1 can find a collision with about $O(2^{61})$ steps;
in theory, this attack *should* take $O(2^{80})$ steps.

As a result, NIST ran a hash function competition to design a replacement for SHA-1 (Keccak chosen as SHA-3 in Oct 2012) **Like the AES competition**

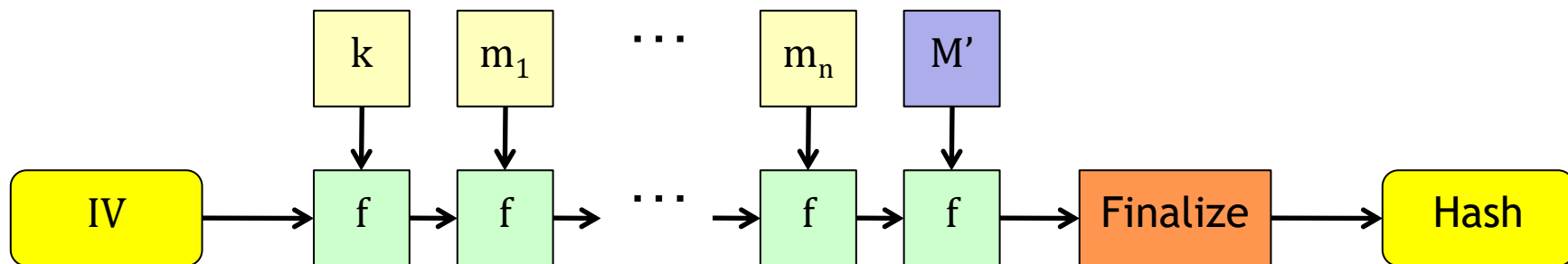
Although hashes are unkeyed functions, they can be used to generate MACs

A keyed hash can be used to detect errors in a message



Unfortunately, this isn't *totally* secure...

It's usually easy to add more data while still generating a correct MAC!



There are also attacks against $H(m || k)$ and $H(k || m || k)$!

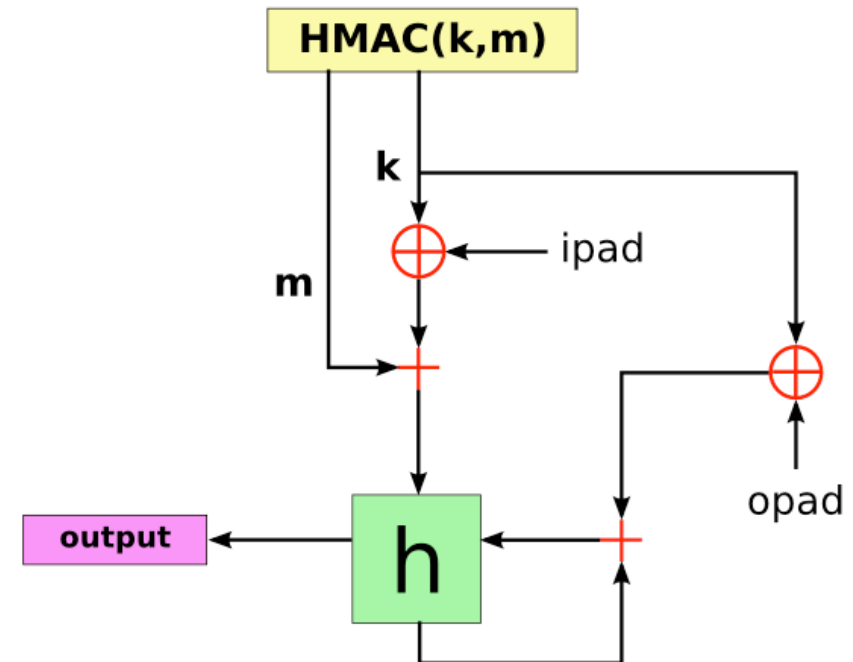
HMAC uses a hash function to generate cryptographically strong MAC

$$\text{HMAC}(k, m) = H((k \oplus \text{opad}) \parallel H((k \oplus \text{ipad}) \parallel m))$$

opad = 01011100 (0x5c)

ipad = 00110110 (0x36)

The opad and ipad constants were carefully chosen to ensure that the internal keys have a large **Hamming distance** between them



Note that H can be **any** hash function. For example, HMAC-SHA-1 is the name of the HMAC function built using the SHA-1 hash function.

Benefits of HMAC:

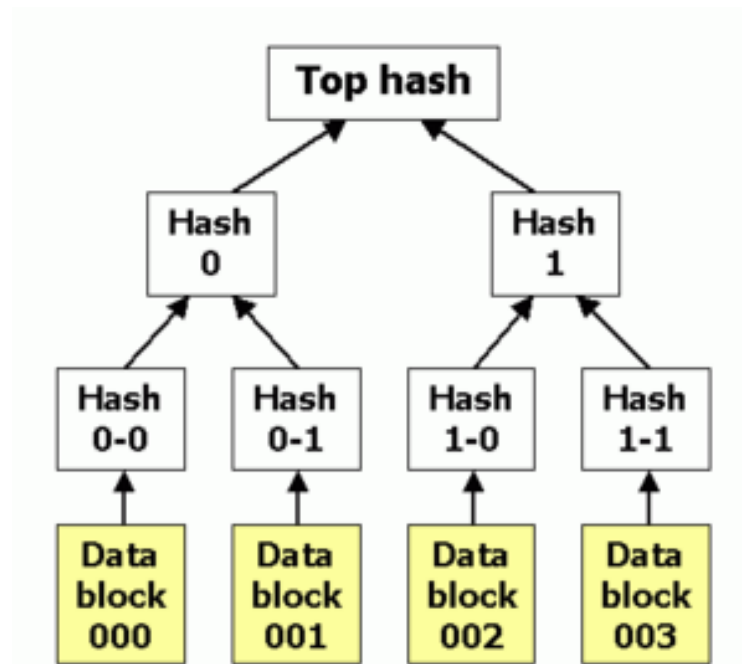
Hash functions are faster than block ciphers

Good security properties

Since HMAC is based on an **unkeyed** primitive, it is not controlled by export restrictions!

Hash functions help check file integrity efficiently

Many peer-to-peer file sharing systems use **Merkle trees** for this purpose



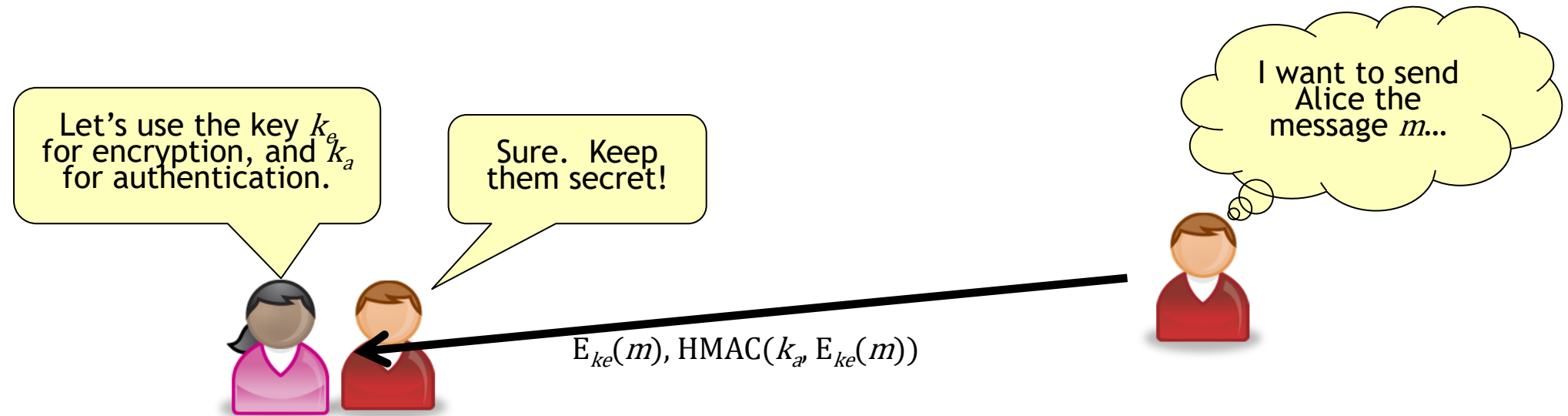
Why is this good?

- One branch of the hash tree can be downloaded and verified at a time
- Interleave integrity check with acquisition of file data
- Errors can be corrected on the fly

BitTorrent uses **hash lists** for file integrity verification

- Must download full hash list prior to verification

Putting it all together...



Why compute the HMAC over $E_{k_e}(m)$?

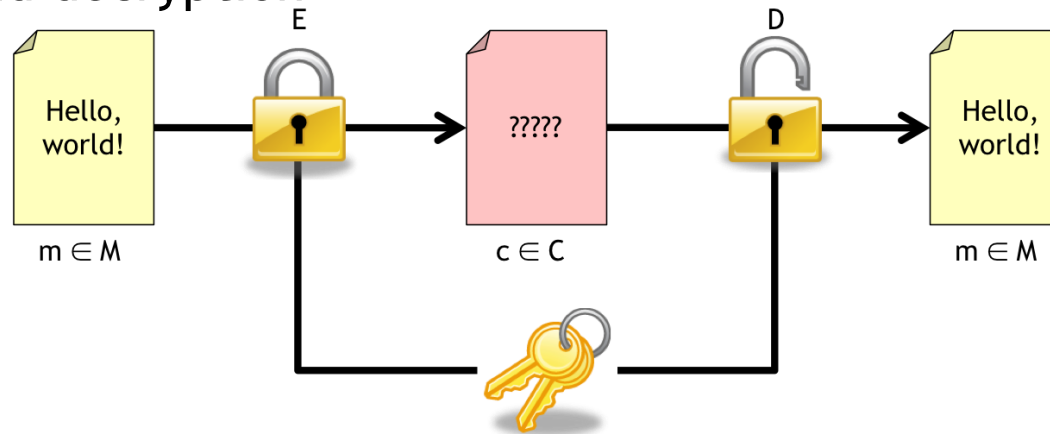
Alice doesn't need to waste time decrypting m if it was mangled in transit, since its authenticity can be checked first!

Why use two separate keys?

In general, it's a bad idea to use cryptographic material for multiple purposes

Symmetric Crypto Pros

Recall: In a symmetric key cryptosystem, the **same** key is used for both encryption and decryption



Note: the sender and recipient need a **shared** secret key

The good news is that symmetric key algorithms

- Have been well-studied by the cryptography community
- Are extremely fast, and thus good for encrypting bulk data
- Provide good security guarantees based on very small secrets

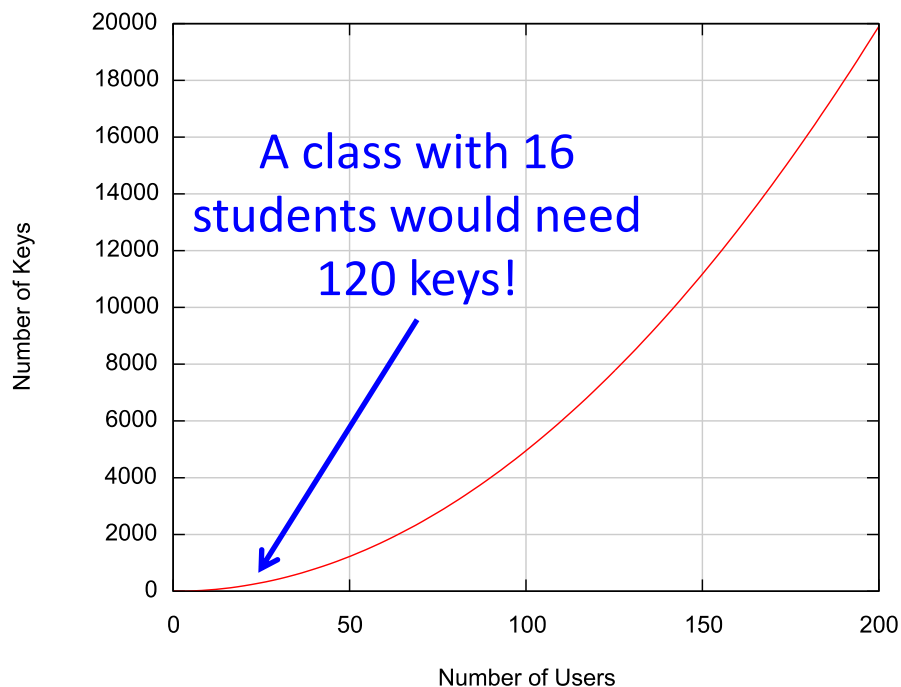
Unfortunately...

Symmetric key cryptography is not a panacea

Question: What are some ways in which the need for a shared secret key might cause a problem?

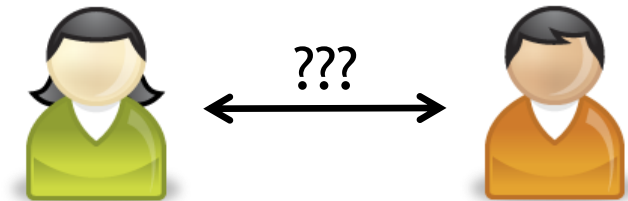
Problem 1: Key management

- In a network with n participants, $\binom{n}{2} = n(n-1)/2$ keys are needed!
- This number grows very rapidly!



Problem 2: Key distribution

- How do Alice and Bob share keys in the first place?



- What if Alice and Bob have never met in person?
- What happens if they suspect that their shared key K_{AB} has been compromised?

Wouldn't it be great if we could securely communicate **without** needing pre-shared secrets?

Thought Experiment

Forget about bits, bytes, ciphers, keys, and math...

The Scenario: Assume that Alice and Bob have never met in person. Alice has a top secret widget that she needs to send to Bob using an untrusted courier service. Alice and Bob can talk over the phone if needed, but are unable to meet in person. Due to the high-security nature of their work, the phones used by Alice and Bob may be wiretapped by other secret agents.

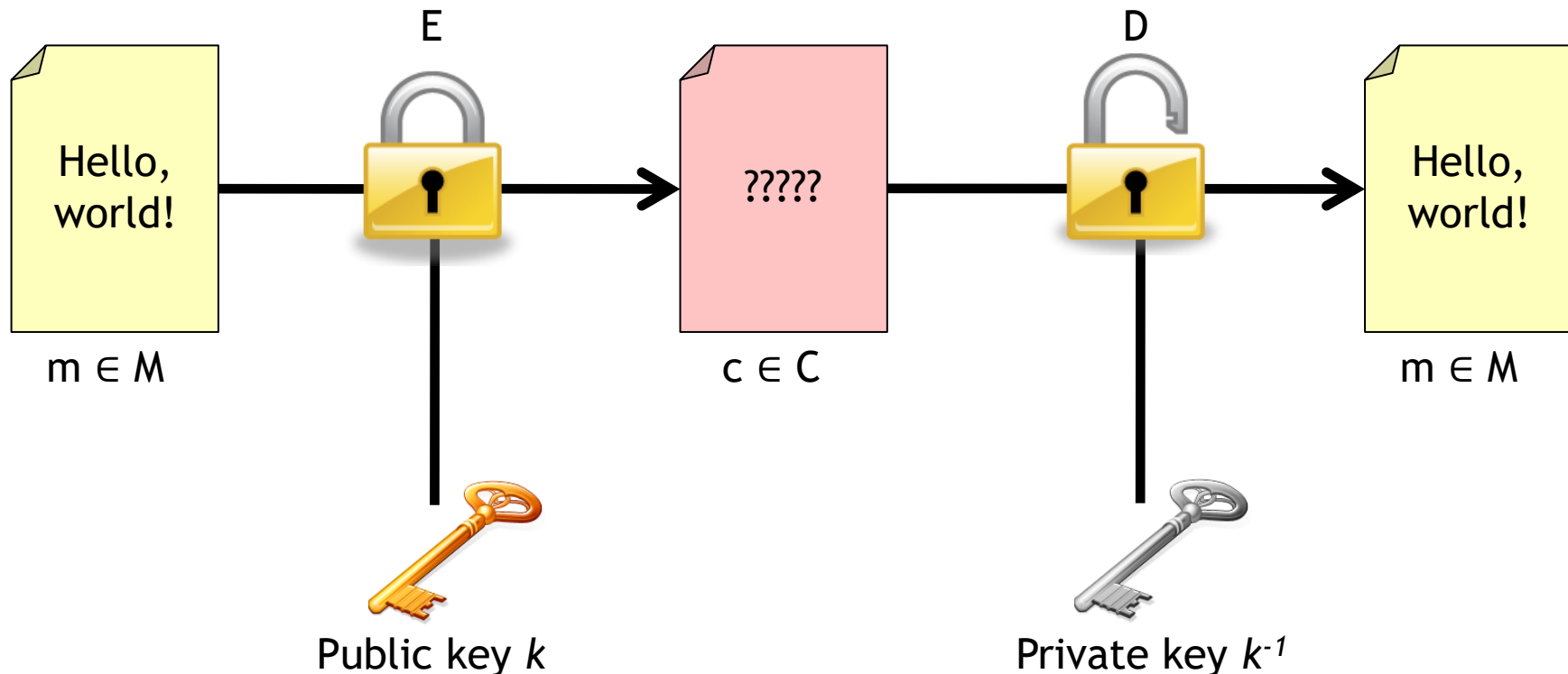
Problem: How can Alice send her widget to Bob while having very high assurance that Bob is the only person who will be able to access the widget if it is properly delivered?

Public key cryptosystems are a digital counterpart to the strongbox example

Formally, a cryptosystem can be represented as the 5-tuple (E, D, M, C, K)

- M is a message space
- K is a key space
- $E : M \times K \rightarrow C$ is an encryption function
- C is a ciphertext space
- $D : C \times K \rightarrow M$ is a decryption function

Note: Each “key” in K is actually a pair of keys, (k, k^{-1})



What can we do with public key cryptography?

First, we need some way of finding a user's public key



Print it in
the newspaper



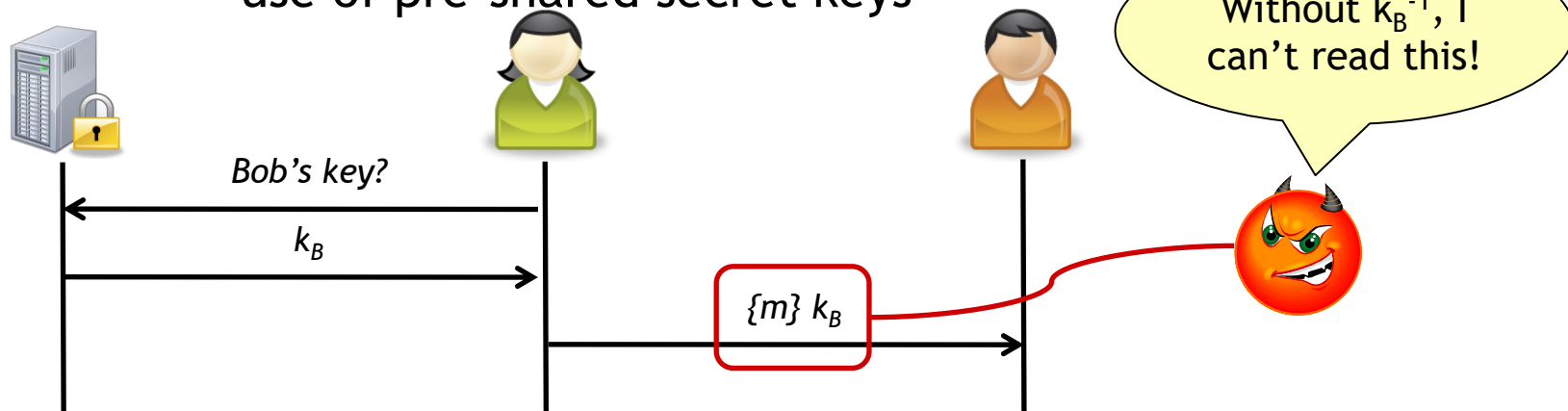
Post it on your
webpage



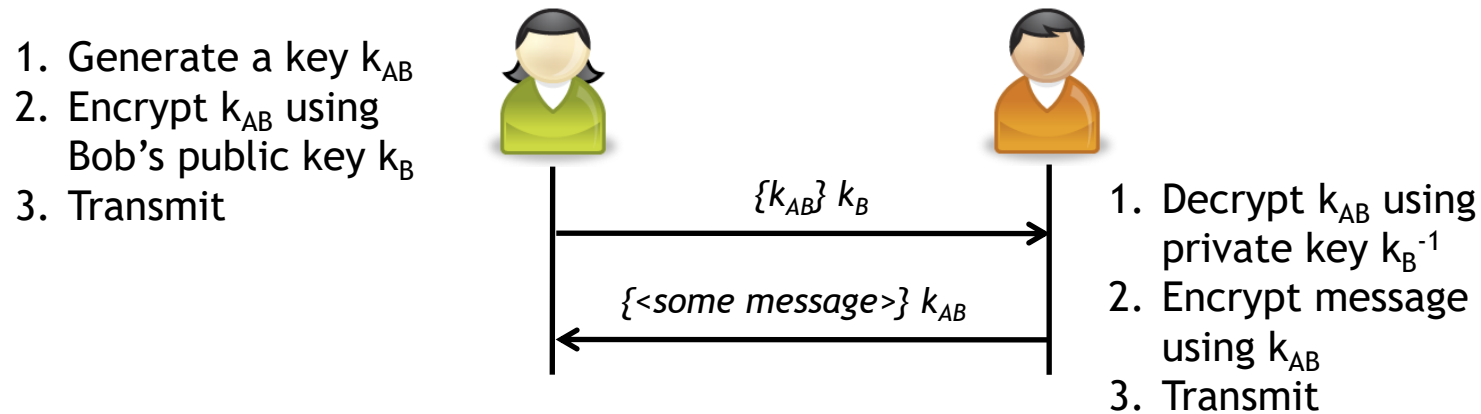
A trusted
keyserver (PKI)

Important: It is critical to verify the authenticity of any public key!
(How?)

Public key cryptography allows us to send private messages **without** the use of pre-shared secret keys



Public key cryptography help exchange symmetric keys



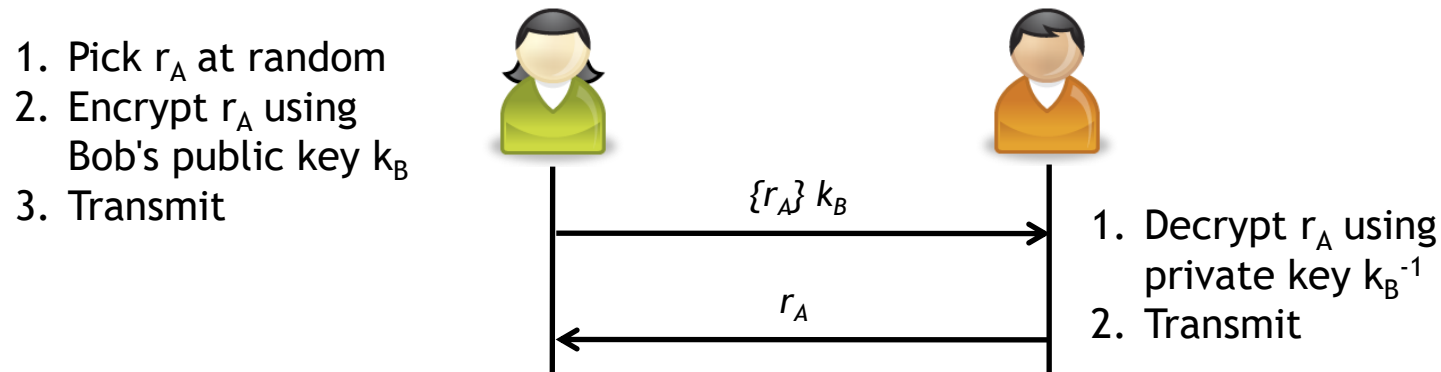
Note: Only Bob can decode k_{AB} , since only he knows k_B^{-1}

- Unfortunately, Bob doesn't know who this key is from
- Key exchange is not quite this easy in practice, but it isn't *much* harder

Question: Why on earth do we want to exchange symmetric keys?!

- Public key cryptography is usually pretty slow...
 - Based on “fancy” math, not bit shifting
 - Symmetric key algorithms are orders of magnitude faster
- It's always a good idea to change keys periodically

Public key cryptography to authenticate users



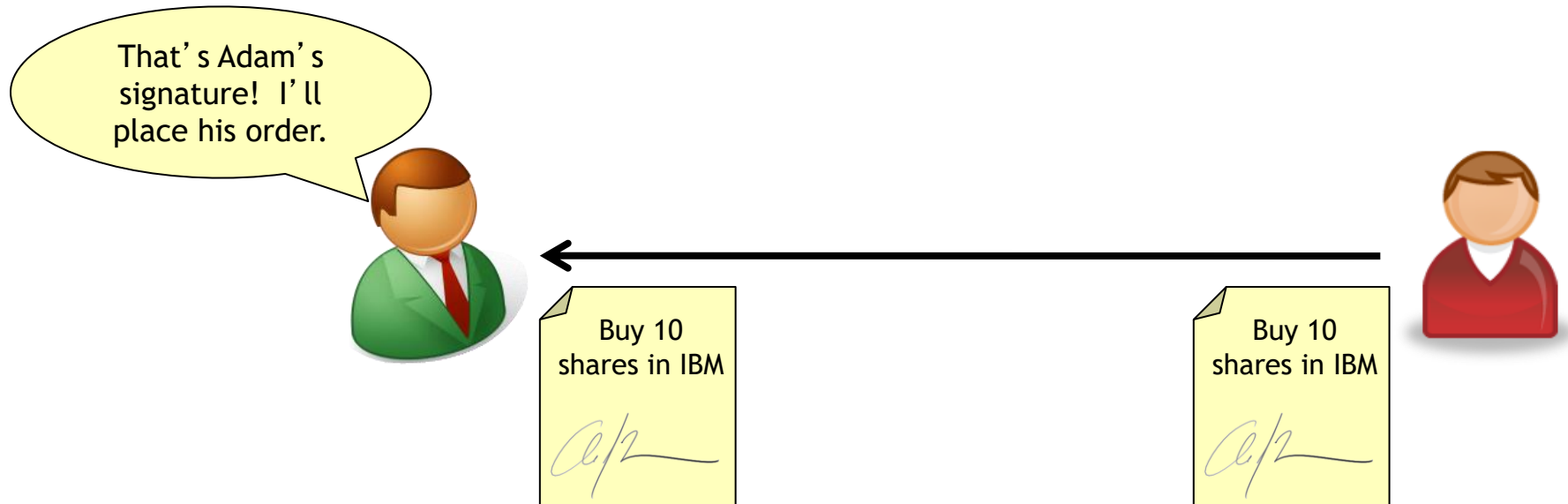
Note: As in the previous key exchange, only Bob can decrypt r_A , since only Bob knows k_B^{-1} .

It is of absolute importance that the random numbers used during this type of protocol are **not predictable** and are **never reused** (Why?)

- **Unpredictable:**
 - The security of this protocol is a proof of possession of k_B^{-1}
 - If predictable, an adversary can guess the “challenge” without decrypting!
 - (This is bad news)
- **Reusing** challenges may* lead to replay attacks (When?)

In addition to encryption, public key systems also let us create digital signatures

Goal: If Bob is given a message m and a signature $S(m)$ supposedly computed by Adam, he can determine whether or not Adam actually wrote the message m



In order for this to occur, we require that

- The signature $S(m)$ must be **unforgeable**
- The signature $S(m)$ must be **verifiable**

Question: *How can we do this?*

In many public key cryptosystems, the encryption and decryption operations are **commutative**

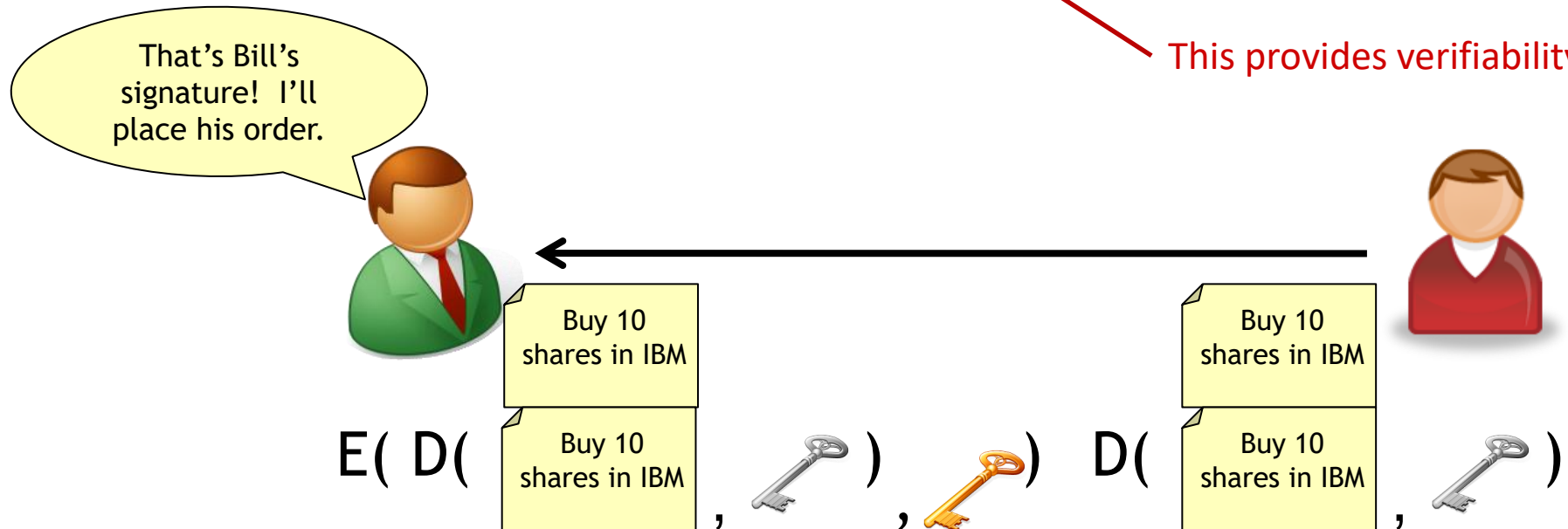
That is, $D(E(m, k), k^{-1}) = E(D(m, k^{-1}), k) = m$

In such a system, we can use digital signatures as follows:

- To sign a message, compute $D(m, k^{-1})$
- Transmit m and $D(m, k^{-1})$ to the recipient
- The recipient uses the sender's public key to verify that $E(D(m, k^{-1}), k) = m$

This is unforgeable

This provides verifiability



Question: Does encryption with a shared key have the same properties?

Features and Requirements

These features all require that for a given key pair (k, k^{-1}) , k can be made public and k^{-1} must remain secret

So, in a public key cryptosystem it must be

1. Computationally **easy** to encipher or decipher a message
2. Computationally **infeasible** to derive the private key from the public key
3. Computationally **infeasible** to determine the private key using a chosen plaintext attack

Informally, **easy** means “polynomial complexity”, while **infeasible** means “no easier than a brute force search”

How do public key cryptosystems work?

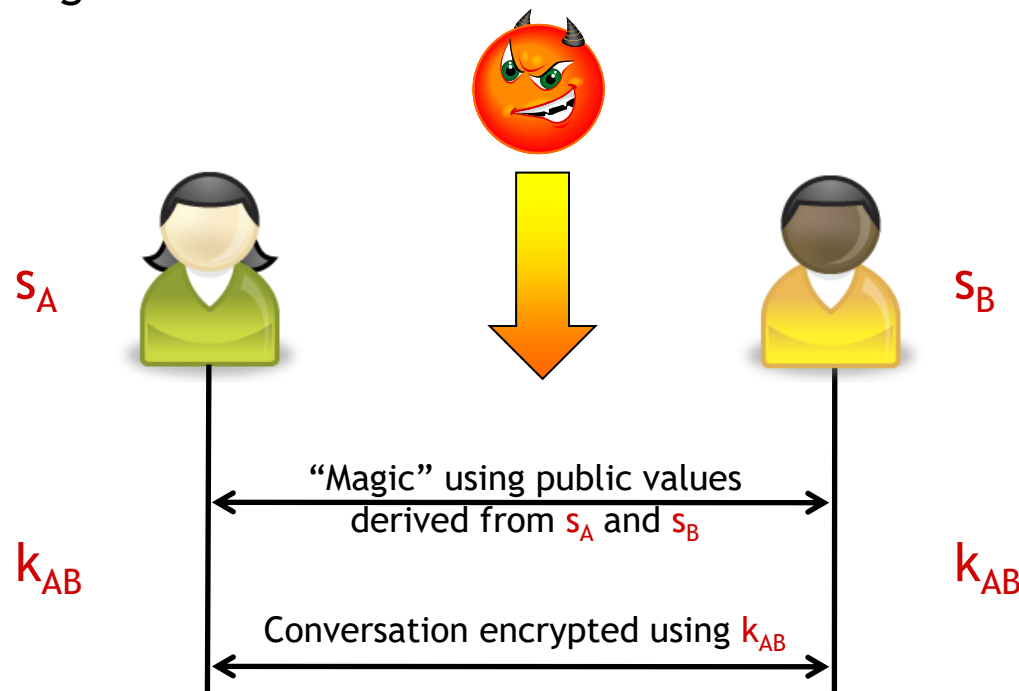
Diffie and Hellman proposed* the notion of public key cryptography

Diffie and Hellman **did not** succeed in developing a full-fledged public key cryptosystem

- i.e., their system cannot be used to encrypt/decrypt document directly
- Rather, it allows two parties to agree on a shared secret using an entirely public channel

Question: Why is this an interesting problem to solve?

- Key exchange!



Diffie and Hellman proposed their system in 1976

Seminal paper: Whitfield Diffie and Martin E. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory (22)6 : 644 - 654, Nov. 1976

Problem: The widening use of telecommunications coupled with the key distribution problems inherent with secret key cryptography point to the fact that current solutions are **not** scalable!

This paper accomplishes many things:

- Clearly articulates why the key distribution problem must be solved
- Motivates the need for digital signatures
- Presents the first public key cryptographic algorithm
- Opened the “challenge” of designing a general purpose public key cryptosystem

Variants of the Diffie-Hellman key exchange algorithm are widely used today!



How does the Diffie-Hellman protocol work?

Step 0: Alice and Bob agree on a finite cyclic group G of (large) prime order q , and a generator g for this group. This information is all **public**.

a is Alice's private key

$g^a \pmod{q}$ is Alice's public key

Step 1:

- Randomly choose $a \in \{1, 2, \dots, q-1\}$
- Compute $g^a \pmod{q}$
- Send $g^a \pmod{q}$



$g^a \pmod{q}$

$g^b \pmod{q}$

Step 2:

- Randomly choose $b \in \{1, 2, \dots, q-1\}$
- Compute $g^b \pmod{q}$
- Send $g^b \pmod{q}$

Step 3:

- Compute $(g^b \pmod{q})^a \pmod{q} = g^{ba} \pmod{q} = K_{ab}$

Step 3':

- Compute $(g^a \pmod{q})^b \pmod{q} = g^{ab} \pmod{q} = K_{ab}$

Why is the Diffie-Hellman key exchange protocol safe?

Recall: We need to show that it is hard for a “bad guy” to learn any of the secret information generated by this protocol, assuming that they know all public information

Public information: $G, g, q, g^a \pmod{q}, g^b \pmod{q}$

Private information: $a, b, K_{ab} = g^{ab} \pmod{q}$

Tactic 1: Can we get $g^{ab} \pmod{q}$ from $g^a \pmod{q}$ and $g^b \pmod{q}$?

- We can get $g^{am+bn} \pmod{q}$ for arbitrary m and n , but this is no help...

Tactic 2: Can we get a from $g^a \pmod{q}$?

- This called taking the discrete logarithm of $g^a \pmod{q}$
- The discrete logarithm problem is widely believed to be very hard to solve in certain types of cyclic groups

Conclusion: If solving the discrete logarithm problem is hard, then the Diffie-Hellman key exchange is secure!

Hm, interesting...

Recall from previous lectures that:

- Block ciphers secure data through confusion and diffusion
 - Designing block cipher mechanisms is equal parts art and science
 - The security of a block cipher is typically accepted over time (**Assurance!**)
 - Recall the initial skepticism over DES
 - The NIST competitions promote this as well
-

In public key cryptography, the relationship between k and k^{-1} is intrinsically mathematical

Result: The security of these systems is also rooted in mathematical relationships, and proofs of security involve reductions to mathematically “hard” problems

- E.g., Diffie-Hellman safe if the discrete logarithm is hard

The RSA cryptosystem picks up where Diffie and Hellman left off

RSA was proposed* by Ron Rivest, Adi Shamir, and Leonard Adelman in 1978. It can be used to encrypt/decrypt and digitally sign arbitrary data!

Key generation:

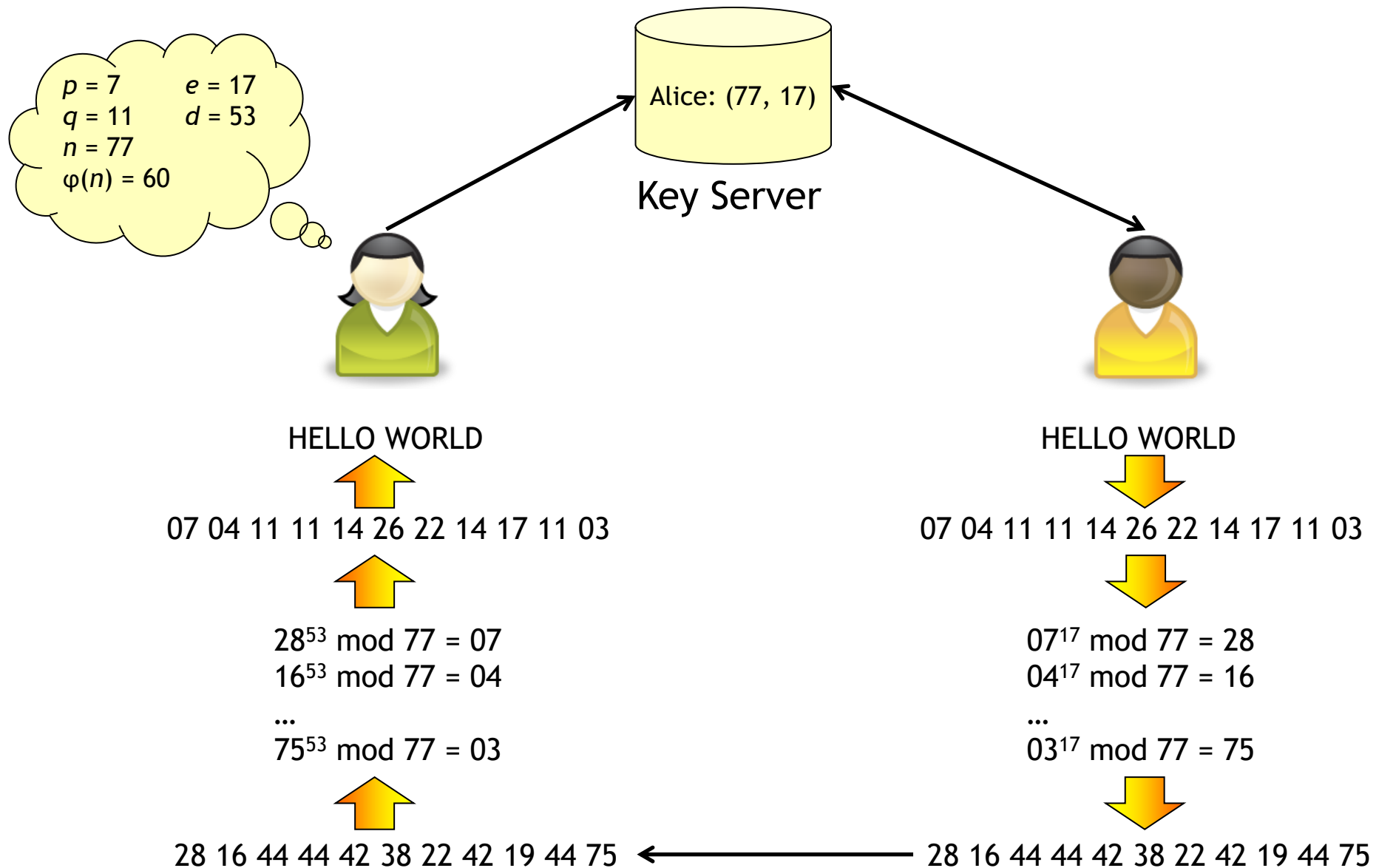
- Choose two large prime numbers p and q , compute $n = pq$
- Compute $\phi(n) = (p-1)(q-1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
- **Public key:** n, e
- **Private key:** p, q, d

We'll discuss how to do these steps and why they work next week

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n) + 1} \pmod{n} = M^1 \pmod{n} = M$

An RSA Example



What is involved in breaking RSA?

To break RSA, an attacker would need to derive the decryption exponent d from the public key (n, e)

Mathematicians think that this is a hard problem

This is **conjectured** to be as hard as factoring n into p and q . Why?

- Given p , q , and e , we can compute $\phi(n)$
- This allows us to compute d easily!

But what if there is some entirely unrelated way to derive d from the public key (n, e) ?

Question: Should this make you uneasy? Why or why not?

My Answer: Probably not, since this bizarre new attack would also have applications to factoring large numbers.

As always, nothing is really that easy...

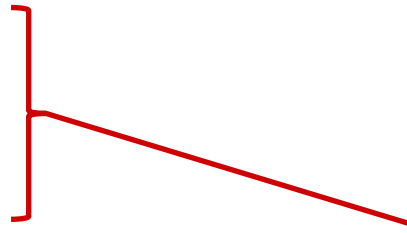
The bad news: Naive implementations of RSA are vulnerable to chosen ciphertext attacks

The good news: These attacks can be prevented by using a padding scheme like OAEP prior to encryption

Don't implement cryptography yourself! Use a standardized implementation, and verify that it is standards compliant.

Lastly, don't forget that implementations can be subjected to attacks

- Timing attacks
- Power consumption attacks
- Etc...

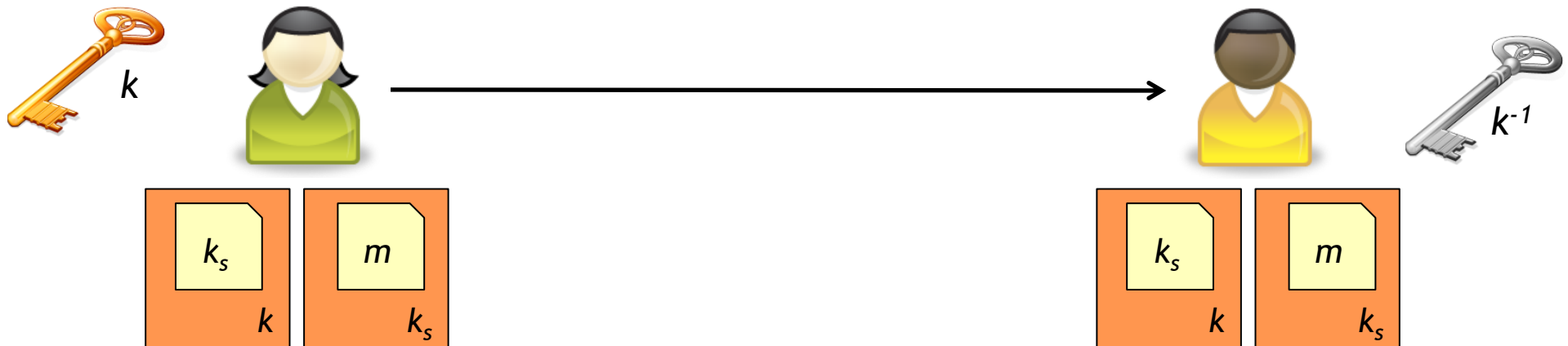


More on this later

Unfortunately, RSA is slow when compared to symmetric key algorithms like AES or HMAC-X

Using RSA as part of a **hybrid cryptosystem** can speed up **encryption**

- Generate a symmetric key k_s
- Encrypt m with k_s
- Use RSA to encrypt k_s using public key k
- Transmit $E_{k_s}(m)$, $E_k(k_s)$




Using hash functions can help speed up **signing** operations

- **Intuition:** $H(m) \ll m$, so signing $H(m)$ takes far less time than signing m
- Why is this safe? H 's **preimage resistance** property!

Some public-key systems have an interesting property known as malleability

Informally, a **malleable** cryptosystem allows meaningful modifications to be made to ciphertexts without revealing the underlying plaintext

- E.g., $E(x) \otimes E(y) = E(x + y)$



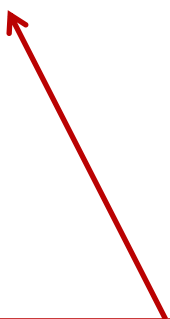
See: Pascal Paillier, Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, EUROCRYPT 1999, pages 223-238.

MacKenzie et al. define a **tag-based** cryptosystem

- Messages encrypted relative to a key and a tag
- Only messages with the same tag can be combined

For example:

- $E(m, t) \otimes E(m', t) = E(mm', t)$
- $E(m, t) \otimes E(m', t') = \text{<garbage>}$



See: Philip MacKenzie, Michael K. Reiter, and Ke Yang, “Alternatives to Non-malleability: Definitions, Constructions, and Applications (Extended Abstract)”, TCC 2004, pages 171-190.

Discussion

Question 1: Why might malleability be an **interesting** property for a cryptosystem to have?

- Tallying electronic votes
- Aggregating private values
- A primitive for privacy-preserving computation
- ...

Question 2: Why might this be **bad**?

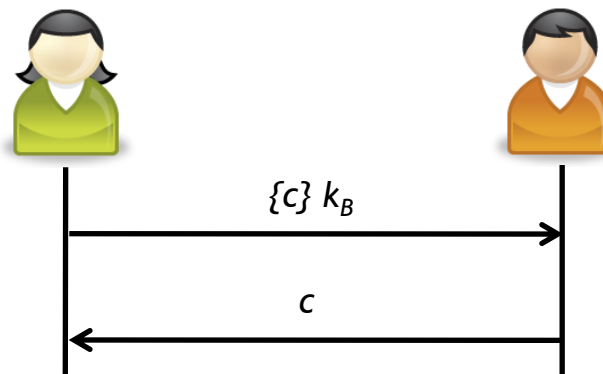
- Modifications by an active attacker!
- **Example:** Modifying an encrypted payment

In short, these types of cryptosystems have interesting properties, but require care to use properly.

Note that public key cryptography allows us to prove knowledge of a secret **without** revealing that secret

Example: Decrypting a challenge

1. Pick challenge c at random
2. Encrypt c using Bob's public key k_B
3. Transmit



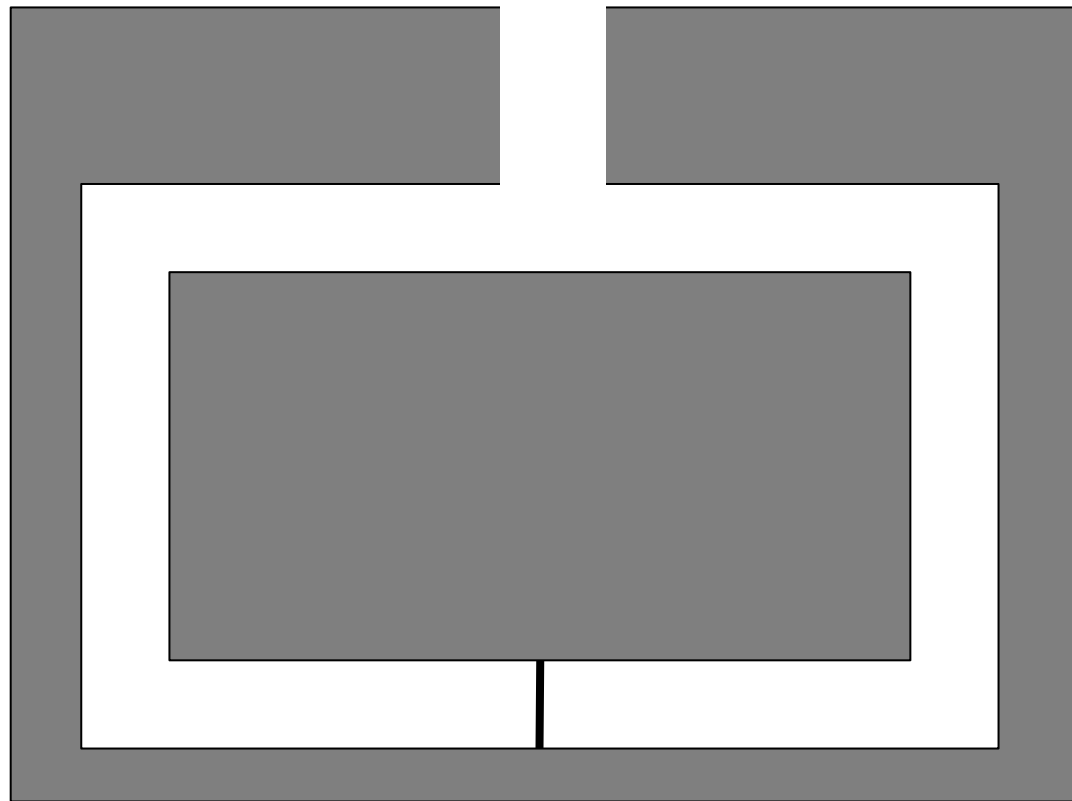
1. Decrypt c using private key k_B^{-1}
2. Transmit

Note: Revealing the challenge, c , does not leak information about the private key k_B^{-1} , yet Alice is (correctly) convinced that Bob knows k_B^{-1}

This type of protocol is called a **zero knowledge** protocol

Zero-knowledge proofs are easy to understand: A children's puzzle

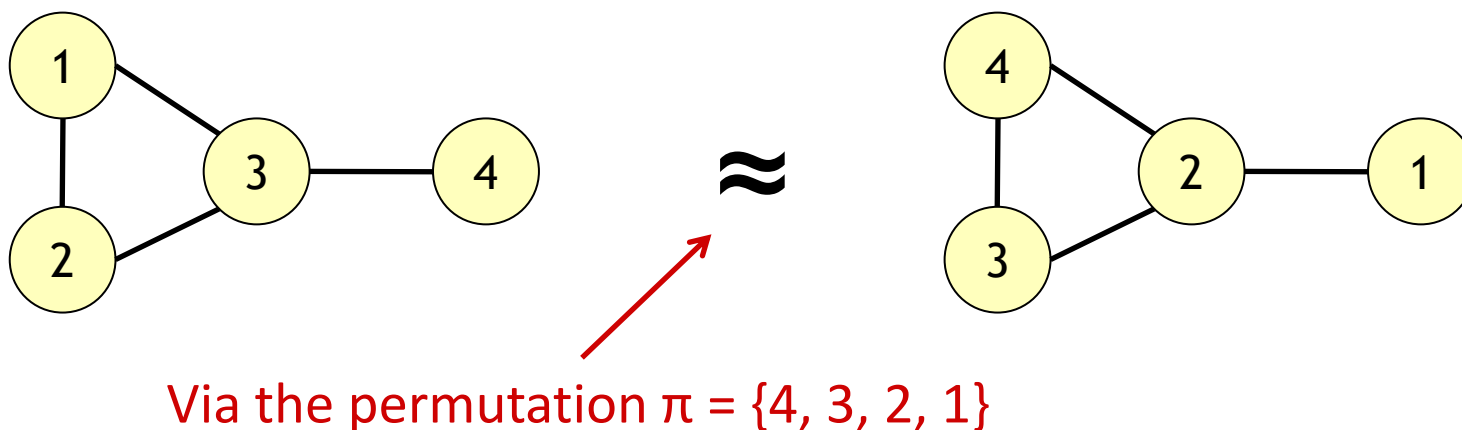
Example: The secret cave



Note: To ensure correctness, this “protocol” needs to be run multiple times (**Why?**)

We can construct a more realistic ZK system based on the (hard) problem of determining graph isomorphism

Informally, two graphs are **isomorphic** if the only difference between them is the names of their nodes



Determining whether two graphs are isomorphic is in NP, but the best known algorithm is $2^{O(\sqrt{n \log n})}$. This means that if the graphs are large, solving this problem will take a long time, but checking a solution is very easy.

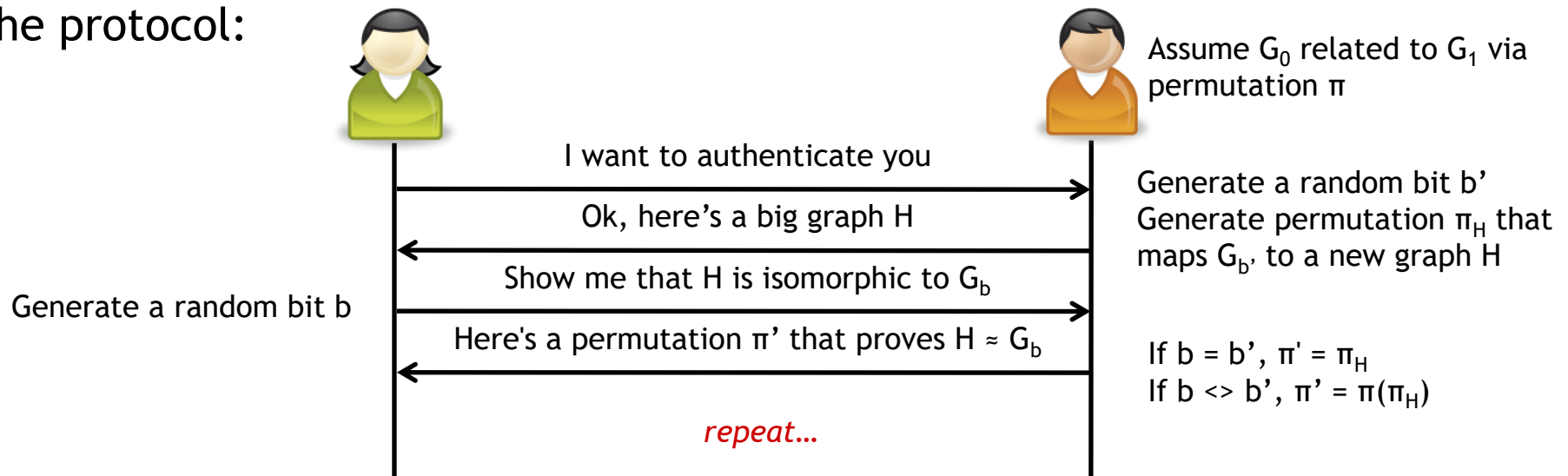
Authenticating via Graph Isomorphism

Our protocol has fairly simple parameters

- **Public key:** Two (big) isomorphic graphs G_0 and G_1
- **Private key:** The permutation mapping $G_0 \rightarrow G_1$

How do we find these efficiently?

The protocol:



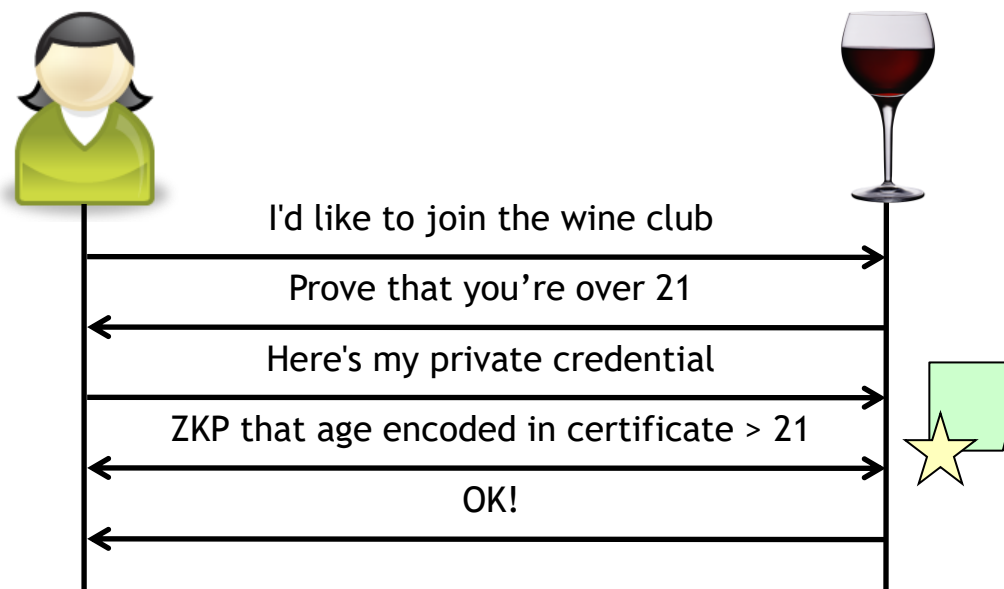
Why does this work?

- Answering this once means that Bob knows (at least) the permutation mapping from G_b to H .
- Doing this m times means that Bob knows the mapping between G_0 and G_1 with probability $1 - 0.5^m$
- Note that this leaks no information regarding the permutation π (Why?)

Zero knowledge proofs of knowledge can be used to solve a variety of interesting **authorization** problems

Private/Anonymous credential systems allow users to prove that they have certain attributes without actually revealing these attributes

Example: Purchasing wine over the Internet



The private credential scheme proposed by Stefan Brands enables many types of attribute properties to be checked in a zero-knowledge fashion

Summary so far ...

Secret key cryptography has a key distribution problem

Public key cryptography overcomes this problem!

- Public encryption key
- Private decryption key

Digital signatures provide both **integrity** protection and **non-repudiation**

Malleable cryptosystems are useful, but their usage entails certain risks

Zero knowledge proof systems have many interesting applications

Next: *Really* understanding RSA

Didn't we learn about RSA?

We saw **what** RSA does and learned a little bit about how we can use those features

Our goal will be to explore

- Why RSA actually works
- Why RSA is efficient* to use
- Why it is reasonably safe to use RSA

In short, it's time for more details ...

Note: Efficiency is a relative term 😊