



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

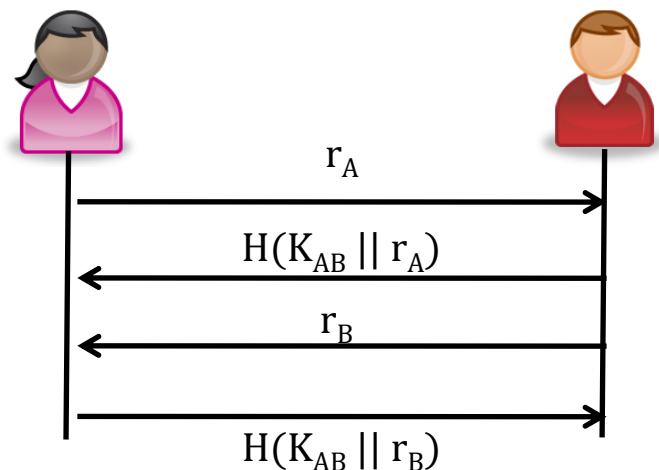
- Homework 3 due this Friday @ 11:59 pm
- Phase 2 of Project
 - posted tonight
 - due on Tuesday 6/27 @ 11:59 pm
- Next week
 - no lecture on Monday
 - Makeup lecture on Friday 6/16 @ 11:00 am

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

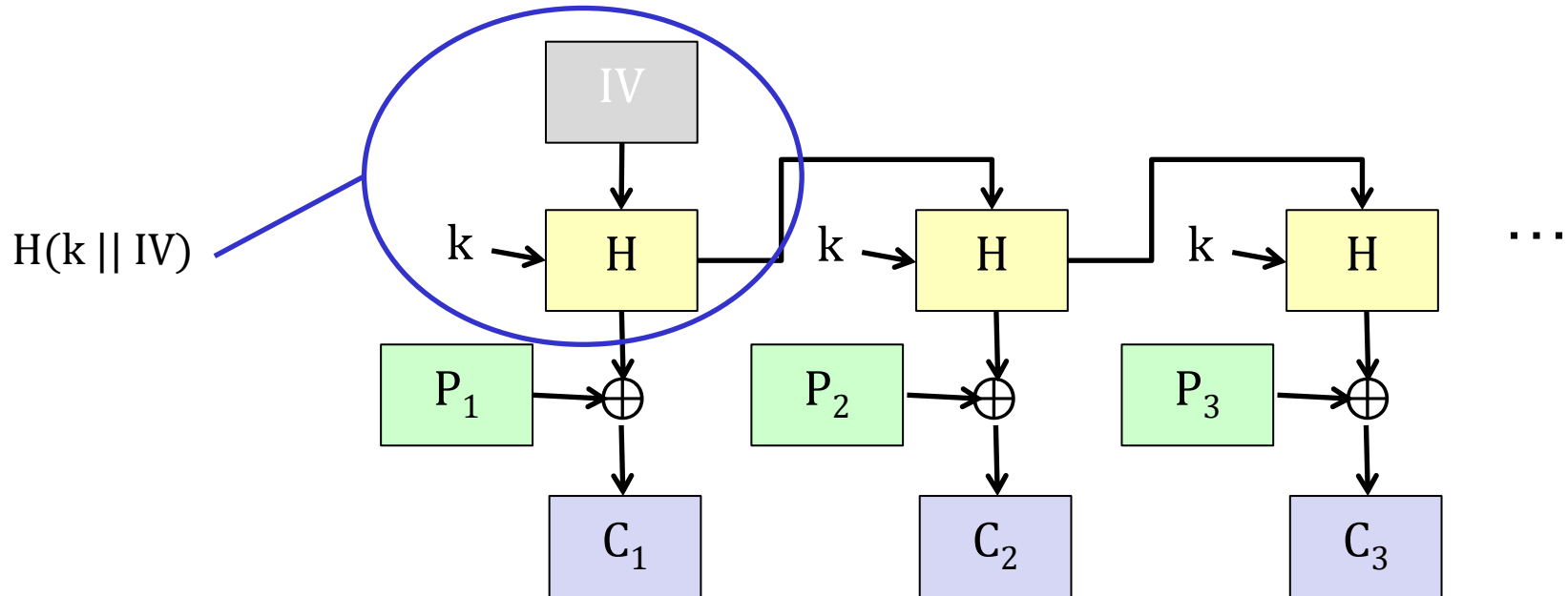
Question: What block cipher mode does this remind you of?

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

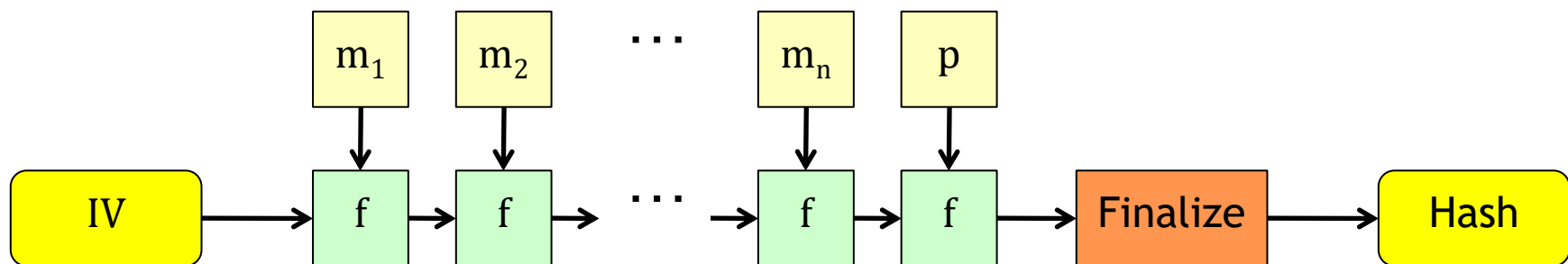
The **Merkle-Damgård construction** is a “template” for constructing cryptographic hash functions

- 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

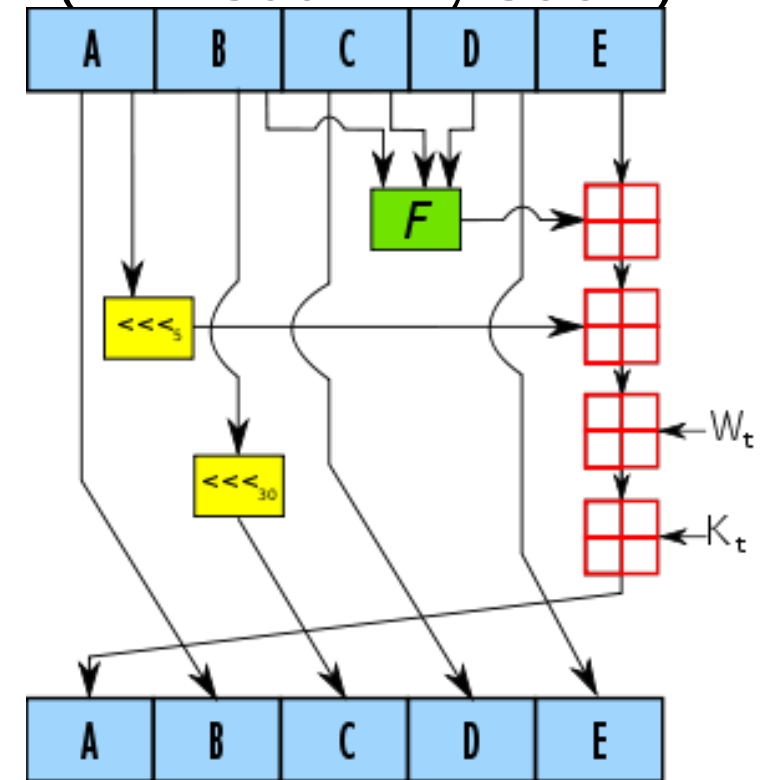


Image from Wikipedia

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:

0xDEADBEEF → 0xDEADBEEF8000 ... 000000020

512 bits

32₁₀ = 0x20

length = 32 bits

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  $0 \leq i \leq 19$  then
```

```
     $f = (b \text{ and } c) \text{ or } ((\text{not } b) \text{ and } d); k = 0x5A827999$ 
```

```
  else if  $20 \leq i \leq 39$ 
```

```
     $f = b \text{ xor } c \text{ xor } d; k = 0x6ED9EBA1$ 
```

```
  else if  $40 \leq i \leq 59$ 
```

```
     $f = (b \text{ and } c) \text{ or } (b \text{ and } d) \text{ or } (c \text{ and } d); k = 0x8F1BBCDC$ 
```

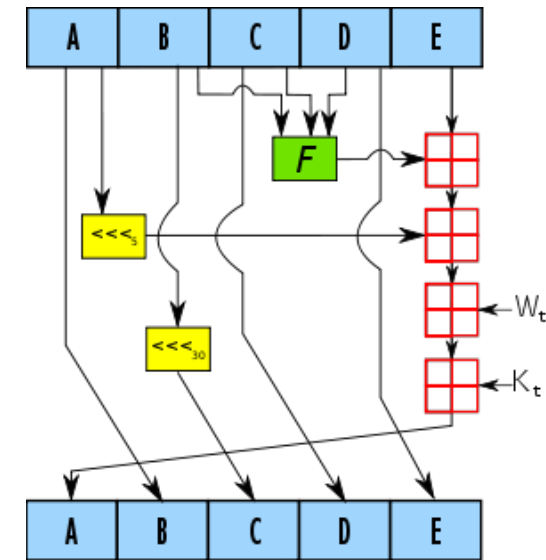
```
  else if  $60 \leq i \leq 79$ 
```

```
     $f = b \text{ xor } c \text{ xor } d; k = 0xCA62C1D6$ 
```

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

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

Note: Sometimes, we treat state as a bit vector...



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

Add this chunk's hash to result so far:

```
h0 = h0 + a; h1 = h1 + b; h2 = h2 + c; h3 = h3 + d; h4 = h4 + 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

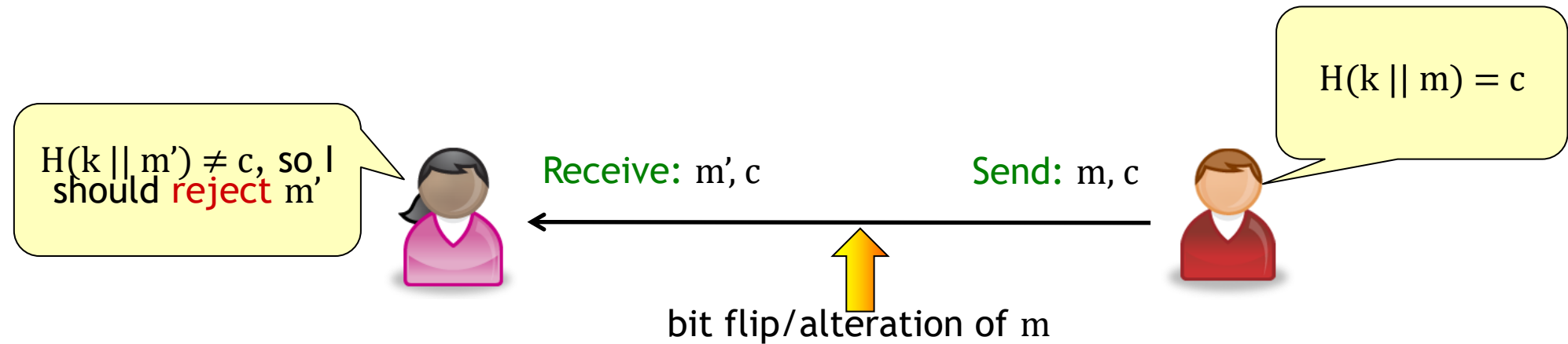
In the above example, changing 1 character of input alters 81 of the 160 bits in the output!

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**

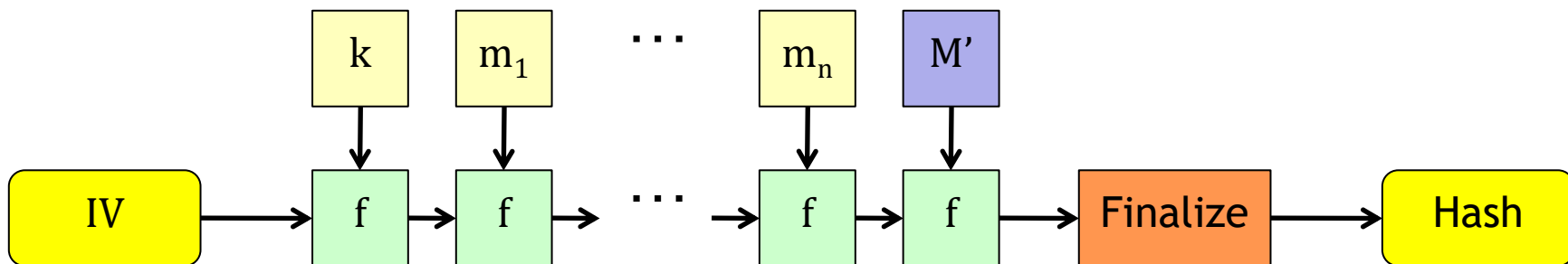
Although hashes are unkeyed functions, they can be used to generate MACs

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 $H(k || m || k)$!

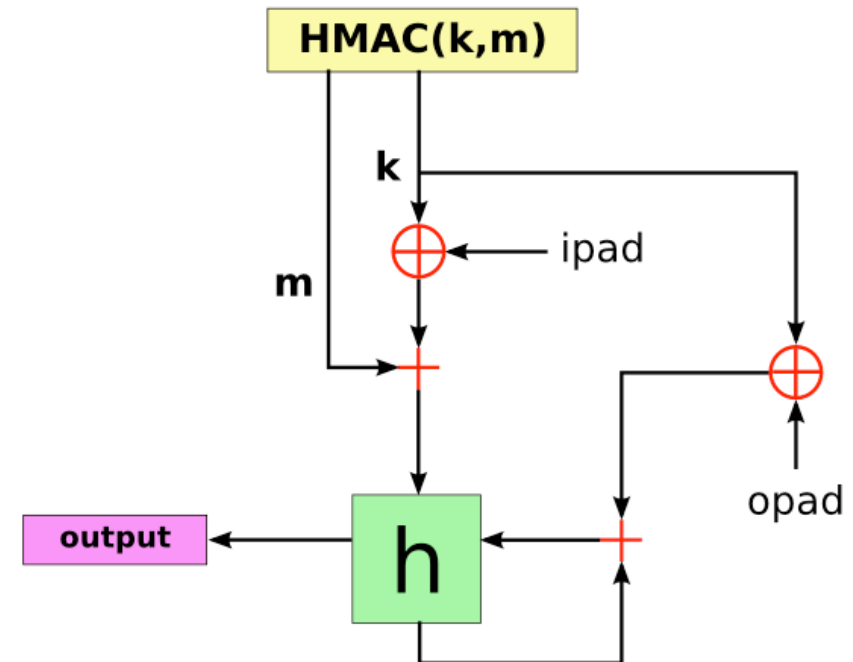
HMAC uses a hash function to generate cryptographically strong MAC

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

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:

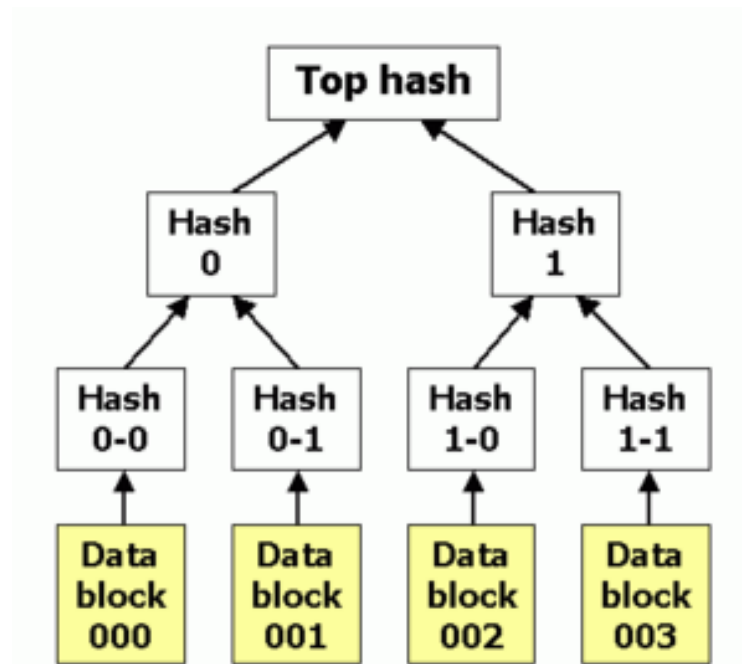
Hash functions are faster than block ciphers

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



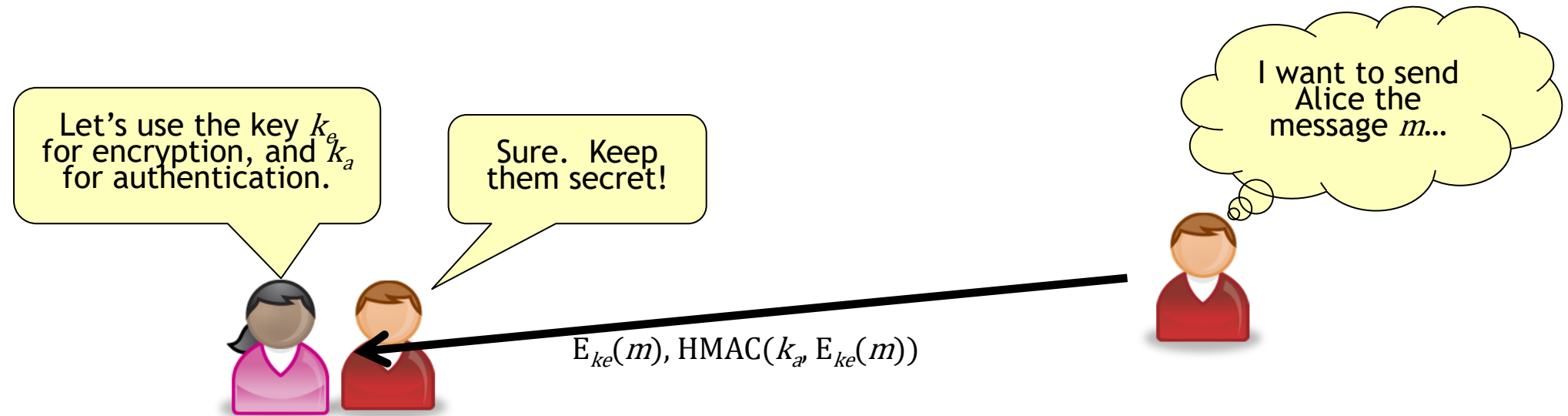
Why is this good?

- 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 $E_{k_e}(m)$?

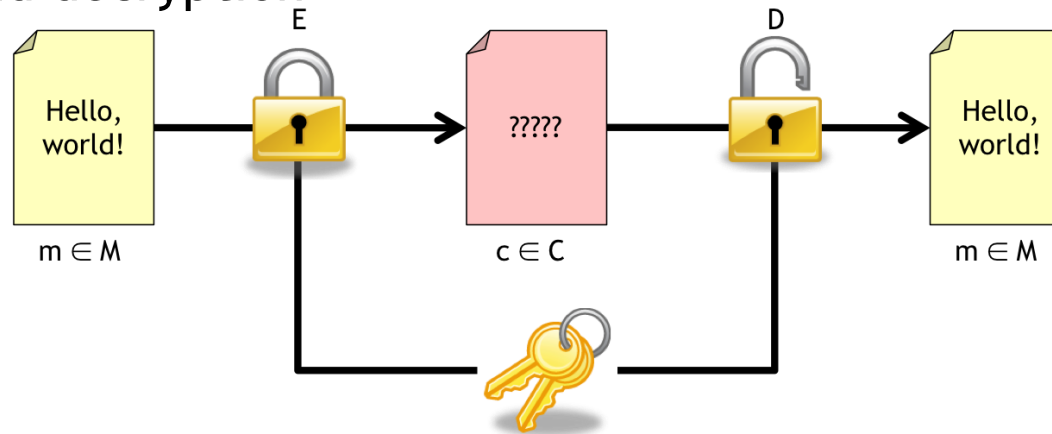
Alice doesn't need to waste time decrypting m if it was mangled in transit, since its authenticity can be checked first!

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

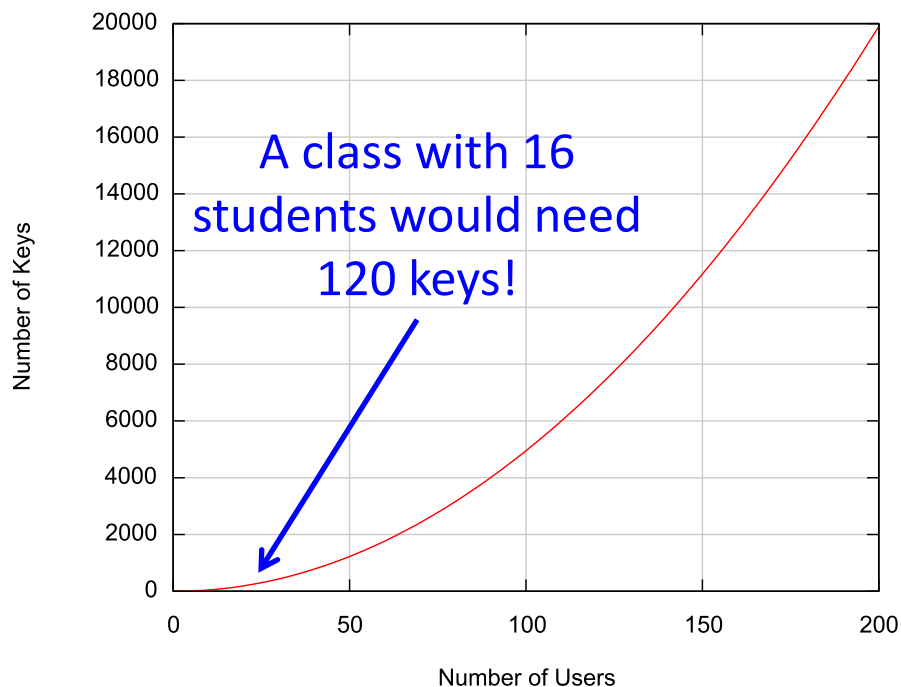
Unfortunately...

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?

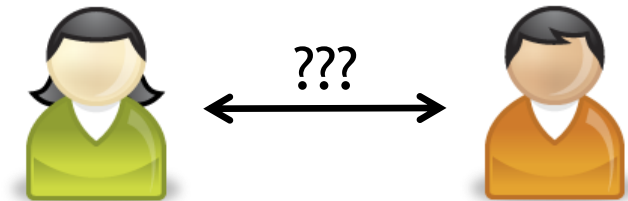
Problem 1: Key management

- 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?