Applied Cryptography and Network Security

Adam J. Lee
adamlee@cs.pitt.edu
6111 Sennott Square

Lecture #6: Hash Functions
January 23, 2014

University of Pittsburgh

What is a hash function?

Definition: A hash function is a function that maps a variable-length input to a fixed-length code

Four score and seven years ago our fathers brought forth on this continent…

0x2bf40a8b690415b69f6f6cc7326597d4

1476 bytes

128 bits

Hash functions are sometimes called message digest functions

- SHA-1 stands for the secure hash algorithm
- MD5 stands for message digest algorithm (version 5)

In order to be useful cryptographically, a hash function needs to have a “randomized” output

For example:

- Given a large number of inputs, any given bit in the corresponding outputs should bet set about half of the time
- Any given output should have half of its bits set on average
- Given two messages m and m’ that are very closely related, H(m) and H(m’) should appear completely uncorrelated

Informally: The output of an *m*-bit hash function should appear as if it was created by flipping *m* unbiased coins

Theoretical cryptographers sometimes use a more formalized notion of random oracles to model hash functions when analyzing security protocols

More formally, cryptographic hash functions should have the following three properties

Assume that we have a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^m$

What does infeasible mean?!?

1. **Preimage resistance:** Given a hash output value z, it should be infeasible to calculate a message x such that $H(x) = z$
   - I.e., H is a one way function
   - Ideally, computing x from z should take $O(2^m)$ time

2. **Second preimage resistance:** Given a message x, it is infeasible to calculate a second message y such that $H(x) = H(y)$
   - Note that this attack is always possible given infinite time (Why?)
   - Ideally, this attack should take $O(2^m)$ time

3. **Collision resistance:** It is infeasible to find two messages x and y such that $H(x) = H(y)$
   - Ideally, this attack should take $O(2^{m/2})$ time

Why only $O(2^{m/2})$?

The Birthday Paradox!

The gist: If there are more than 23 people in a room, there is a better than 50% chance that two people have the same birthday

Wait, what?

- 366 possible birthdays
- To solve: Find probability p_n that n people all have *different* birthdays, then compute $1-p_n$

$$p_n = \frac{365}{366}\frac{364}{366}\frac{363}{366}\cdots\frac{367-n}{366}$$

- If $n = 22$, $1 - p_n \approx 0.475$
- If $n = 23$, $1 - p_n \approx 0.506$

Note: The value of n can be approximated as $1.1774 \times \sqrt{n} = 1.1774 \times \sqrt{366} \approx 22.525$

What the heck does this have to do with hash functions?!

Note that "birthday" is just a function b : person → date

Goal: How many inputs x to the function b do we need to consider to find x_i, x_j such that $b(x_i) = b(x_j)$?

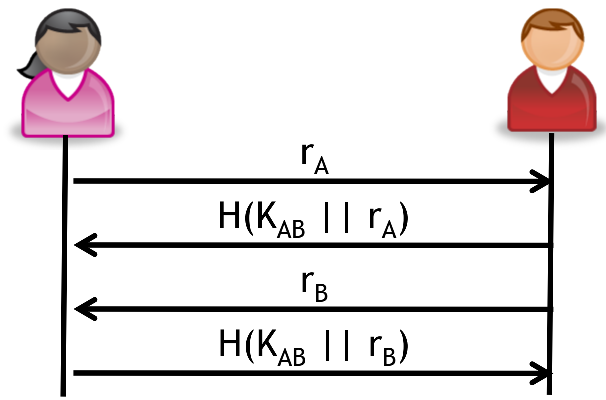
We're looking for collisions in the birthday function!

Now, a hash is a function $H : \{0, 1\}^* \rightarrow \{0, 1\}^m$

- Note: H has 2^m possible outputs

So, using our approximation from the last slide, we'd need to examine about $1.1774 \times \sqrt{2^m} = 1.1774 \times 2^{m/2} = O(2^{m/2})$ inputs to find a collision!

What are some things that we can do with a hash function?



Mutual Authentication

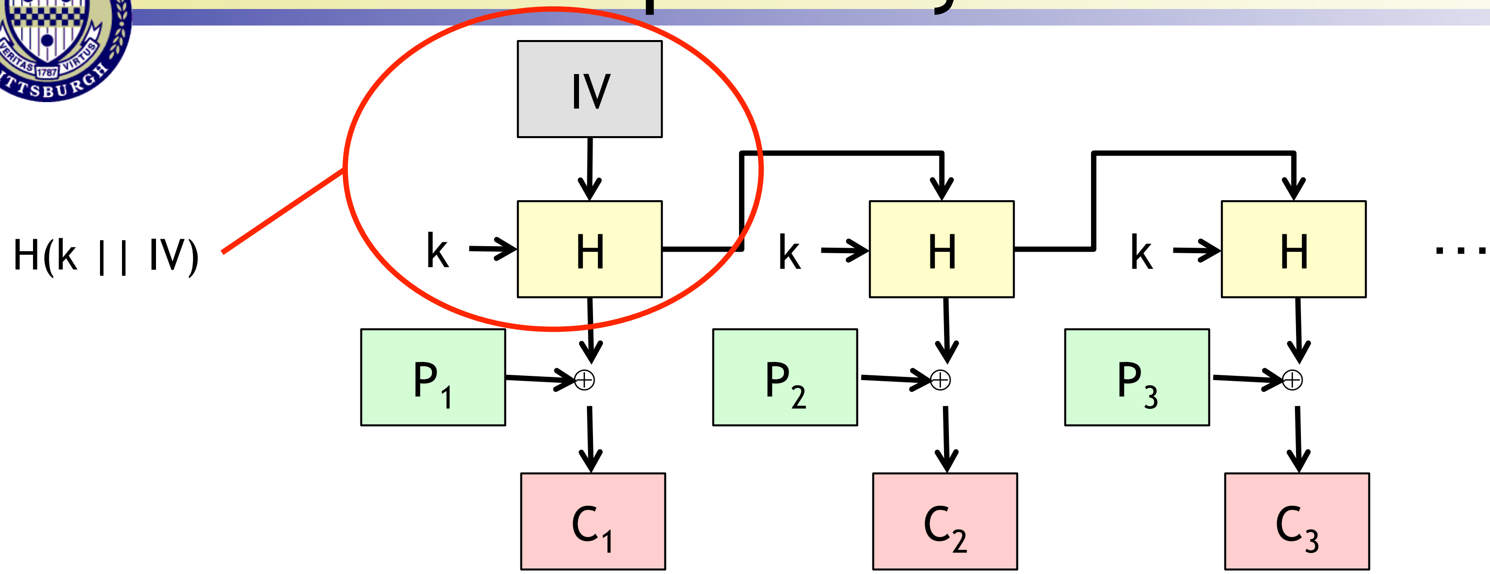
Document Fingerprinting

- Use H(D) to see if D has been modified
- ***Example:*** Tripwire

MAC Functions

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

Hash functions can even be used to generate cipher keystreams



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

- Output feedback mode (OFB), of course!

Why is this safe to do?

- Remember that hash functions need to have behave "randomly" in order to be used in cryptographic applications
- Even if the adversary knows the IV, he 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 ⟨username, password⟩ 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 ⟨username, H(password)⟩ pairs on disk

- Correctness:
  - Host computes H(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 previous applications provide us with an intuitive way to understand the importance of a 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)*

In lecture 16, we'll see that this can lead to attacks that let us inject arbitrary content into protected documents!

Ok, enough high-level talk. How do these things 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 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 in Lecture 16

Rather than discuss MD5, we'll focus on SHA-1

SHA-1 is built using the Merkle-Damgård construction

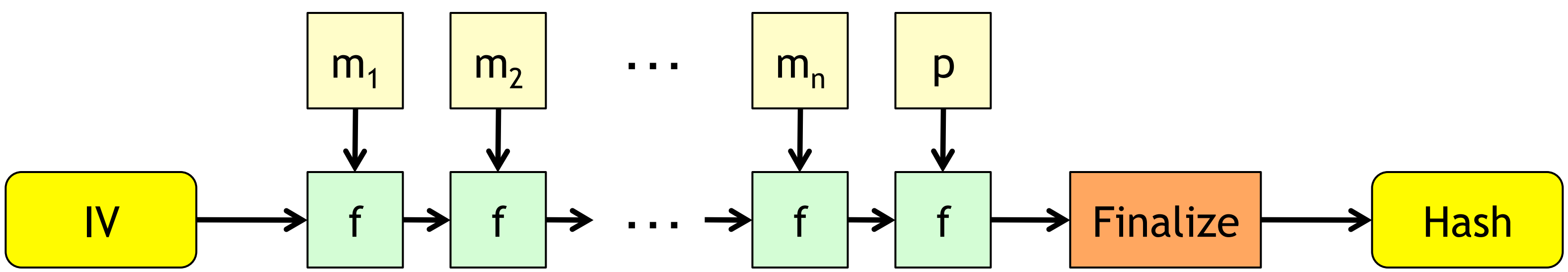
The Merkle-Damgård construction is a “template” for constructing cryptographic hash functions

- 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?



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

A thousand-mile view…

Input: A message of bit length ≤ $2^{64} - 1$

Output: A 160-bit digest

Steps:

- Pad message to a multiple of 512 bits
- Process one 512 bit chunk at a time
- Expand the sixteen 32-bit words into eighty 32-bit words
- Initialize five 32-bit words of state
- For each block of five 32-bit words
  - ➢ Apply function at right
  - ➢ Add result to output
- Concatenate five 32-bit words of output state

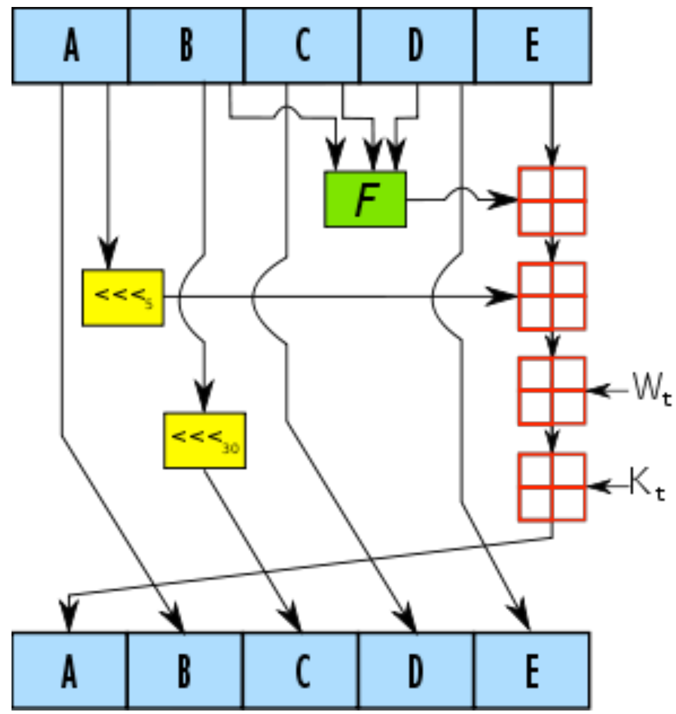


Image from Wikipedia

Initialization and Padding

Initialize variables:

h0 = 0x67452301

h1 = 0xEFCDAB89

h2 = 0x98BADCFE

h3 = 0x10325476

h4 = 0xC3D2E1F0

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

Pre-processing:

append the bit '1' to the message

append $0 \le k < 512$ bits '0', 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

Example:

0xDEADBEEF → 0xDEADBEEF8000 ... 0020

32 bits

32_{10} = 0x20

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], $0 \le i \le 15$

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

    for i from 16 to 79

        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) <<< 1

    Initialize hash value for this chunk:

    a = h0

    b = h1

    c = h2

    d = h3

    e = h4

Note: <<< denotes a left rotate.

Example: 00011000 <<< 4

$\Downarrow$

10000001

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

    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!

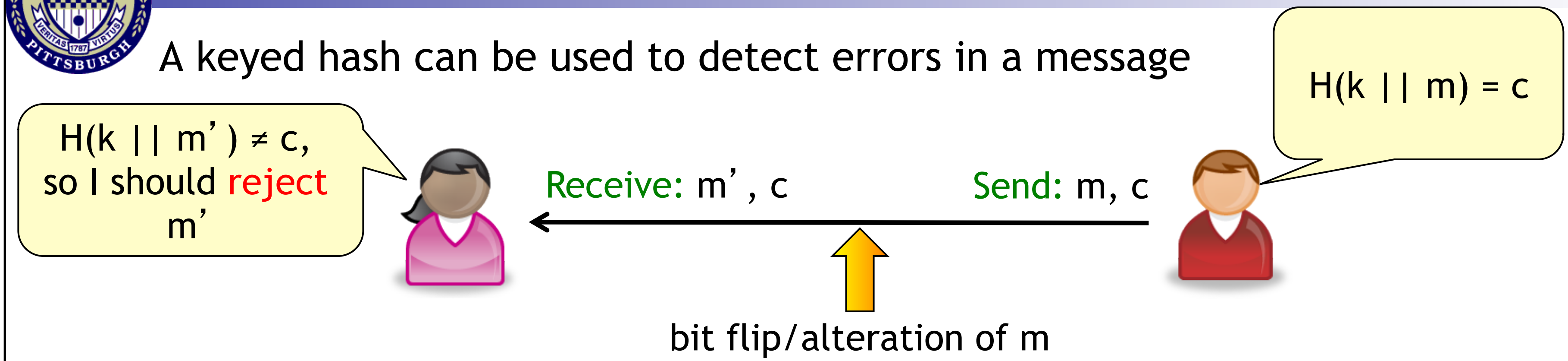
To date, the best attack on SHA-1 can find a collision with about $O(2^{61})$ steps; in theory, this attack *should* take $O(2^{80})$ steps.

As a result, NIST ran a hash function competition to design a replacement for SHA-1 (Keccak chosen in Oct 2012)

Like the AES competition

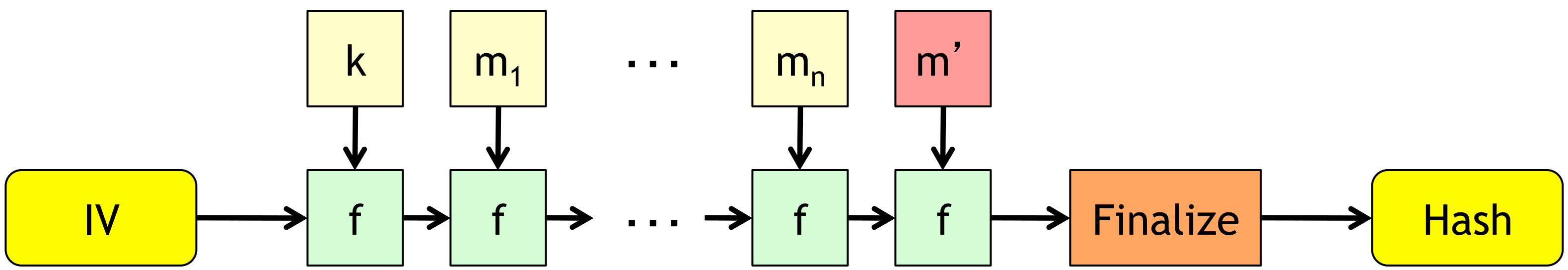
Although hashes are unkeyed functions, they can be used to generate MACs

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



Unfortunately, this isn’t *totally* secure… (Why?)

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

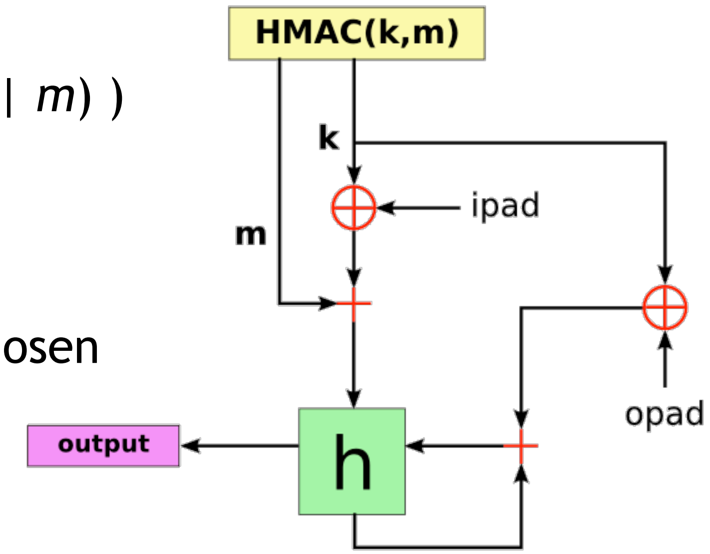


There are also attacks against H(m || k) and H(k || m || k)!

HMAC is a construction that uses a hash function to generate a cryptographically strong MAC

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

- opad = 01011010
- ipad = 00110110



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

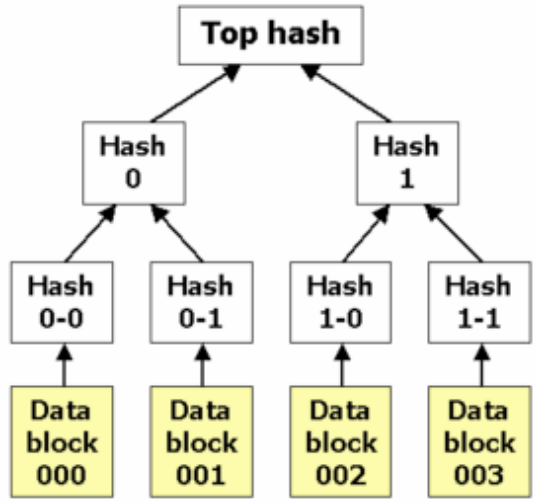
Note that H can be any hash function. For example, HMAC-SHA-1 is the name of the HMAC function built using the SHA-1 hash function.

Benefits of HMAC:

- Hash functions are faster than block ciphers
- Good security properties (Why?)
- Since HMAC is based on an unkeyed primitive, it is not controlled by export restrictions!

Hash functions can also help us check the integrity of large files efficiently

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



Why is this good?

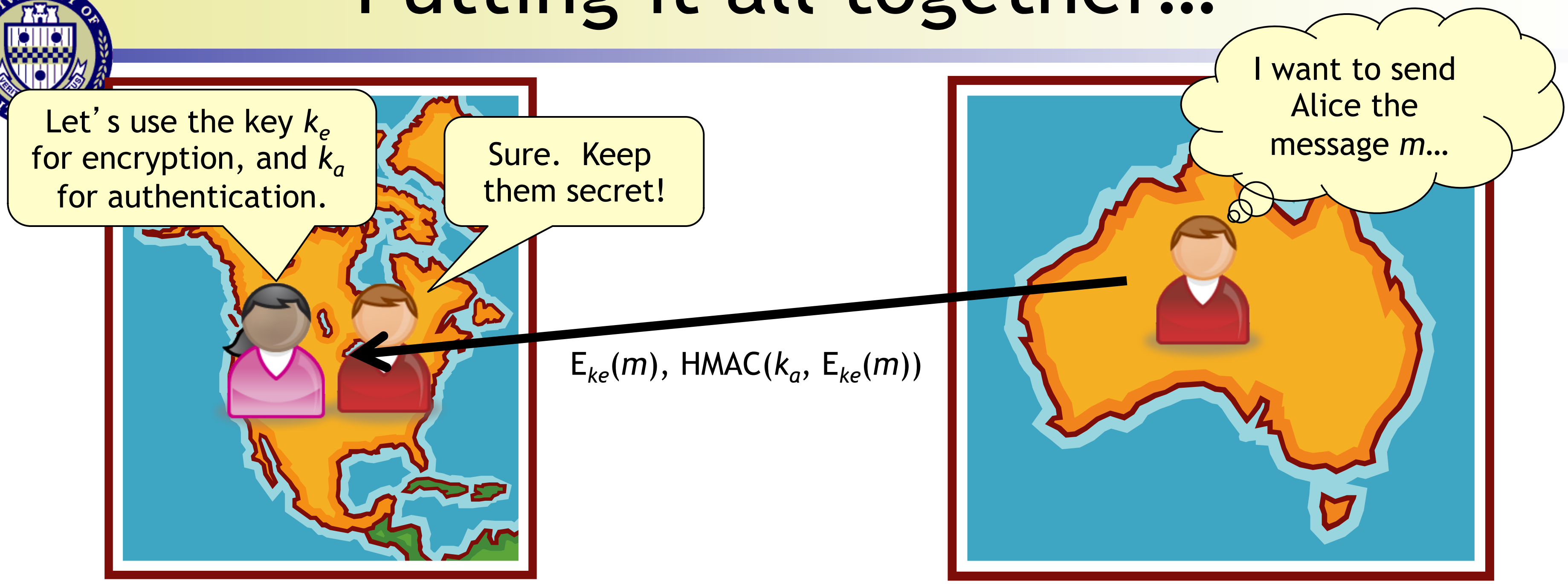
- One branch of the hash tree can be downloaded and verified at a time
- Interleave acquisition of integrity check with file data
- Errors can be corrected on the fly

BitTorrent uses hash lists for file integrity verification

- Must download full hash list prior to verification

Image from Wikipedia

Putting it all together…



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

- 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

Project #2