Applied Cryptography and Network Security

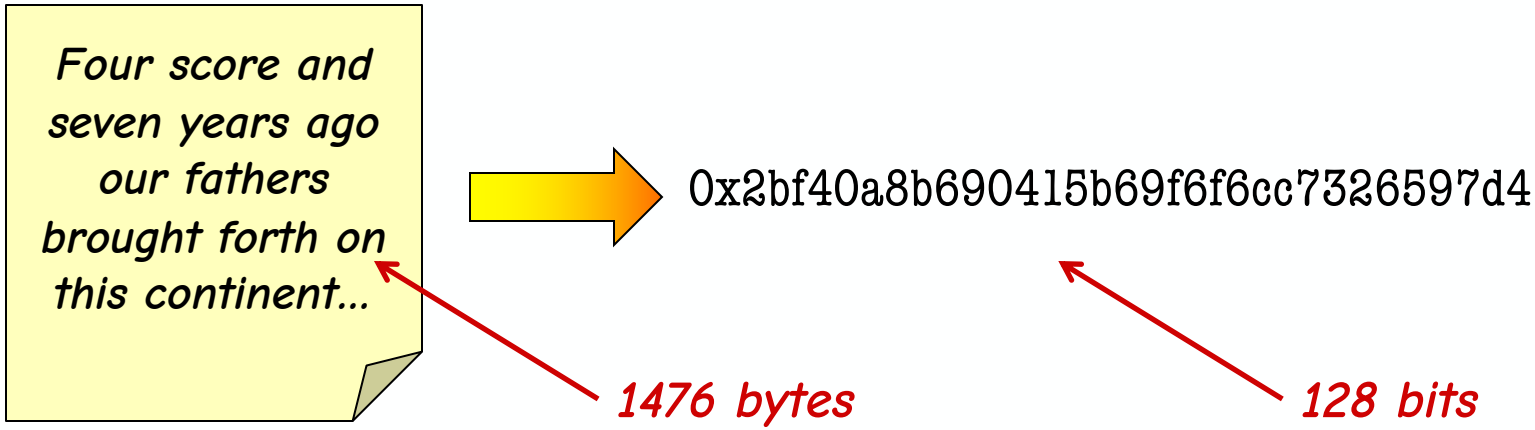
William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Hash Functions and Applications

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



Hash functions are sometimes called message digest functions

- SHA (e.g., SHA-1, SHA-256, SHA-3) 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, $\mathrm{H}(m)$ and $\mathrm{H}(m')$ 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 ("looks random")

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 $1.1774 \times \sqrt{N} = 1.1774\, x\, \sqrt{366} \approx 22.525$

What does this have to do with hash functions?

Note that "birthday" is just a mapping b: person $\rightarrow$ date

Goal: How many inputs to the function b are needed to find x_i, x_j such that $b(x_i) = b(x_j)$?

We're looking for collisions in the birthday function!

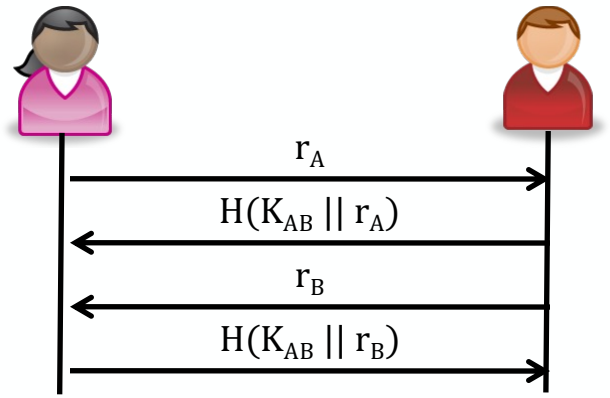
A hash function is $H: \{0,1\}^* \rightarrow \{0,1\}^m$

- Note: H has 2^m possible outputs

Generating n outputs produces $\frac{n(n-1)}{2}$ pairs, each of which has $\frac{1}{2^m}$ chance of colliding

- $\frac{n(n-1)}{2} * \frac{1}{2^m} = \frac{1}{2}$ means $n \approx \sqrt{2^m} = 2^{\frac{m}{2}}$

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



Mutual Authentication

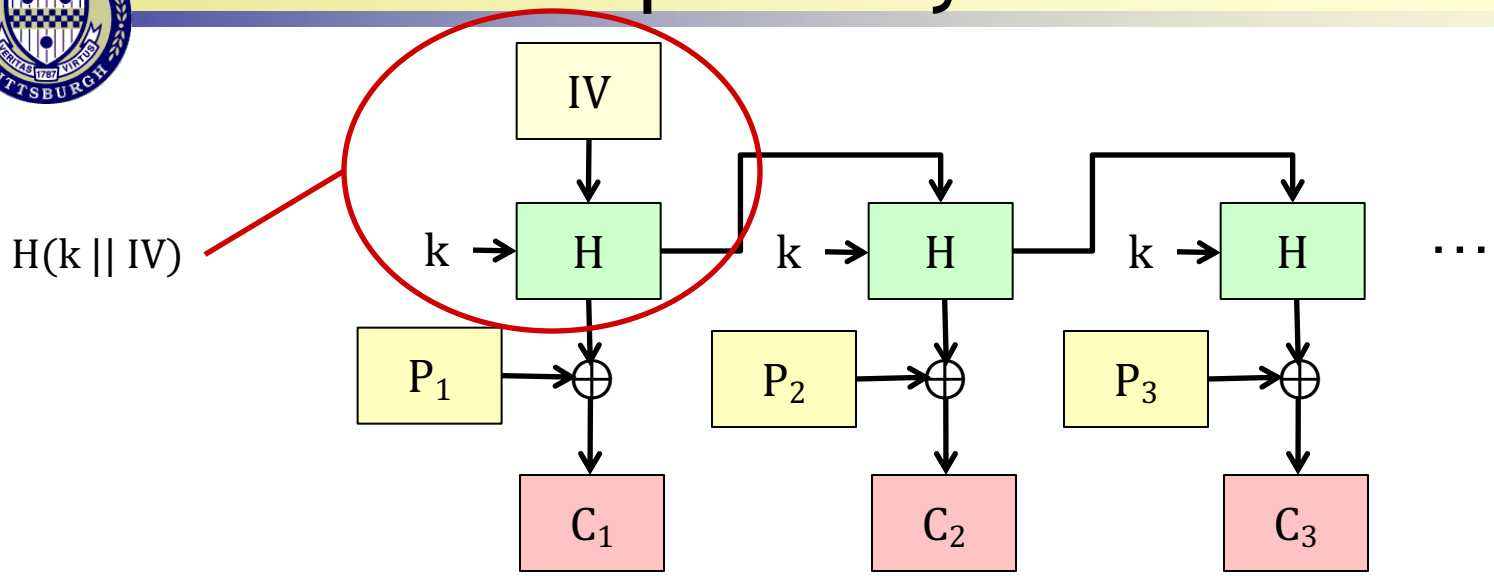
Document Fingerprinting

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

MAC Functions

- 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 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

In practice, storing passwords is a bit more complex (more later)

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 $\mathrm{H}(x) = z$

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 $\mathrm{H}(x) = \mathrm{H}(y)$

- ***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 $\mathrm{H}(x) = \mathrm{H}(y)$

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

The search for cryptographic hash functions was inspired by public-key cryptography

As we will see, public-key systems can produce digital signatures

- (Can only be generated using a private key, but can be verified by anyone with a corresponding public key)
- … but they are also very slow compared with block ciphers

Rather than signing a large message (very expensive), we sign its hash

- Anyone can:
  - hash the message to acquire the digest
  - Use the public key to verify that the signature matches the digest
- Cryptographic properties ensure that no other message can be found with the same digest (and thus the same signature)

Many hash functions are based on the Merkle-Damgård construction

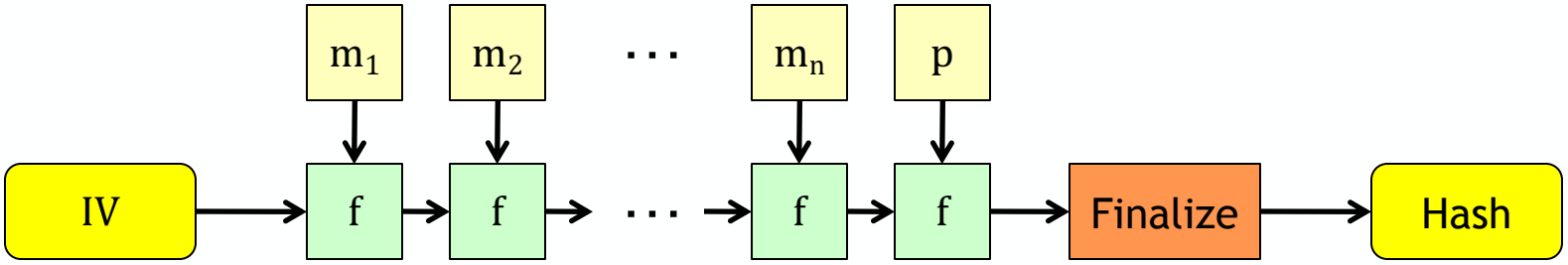
The Merkle-Damgård construction is a "template" for constructing cryptographic hash functions

- 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

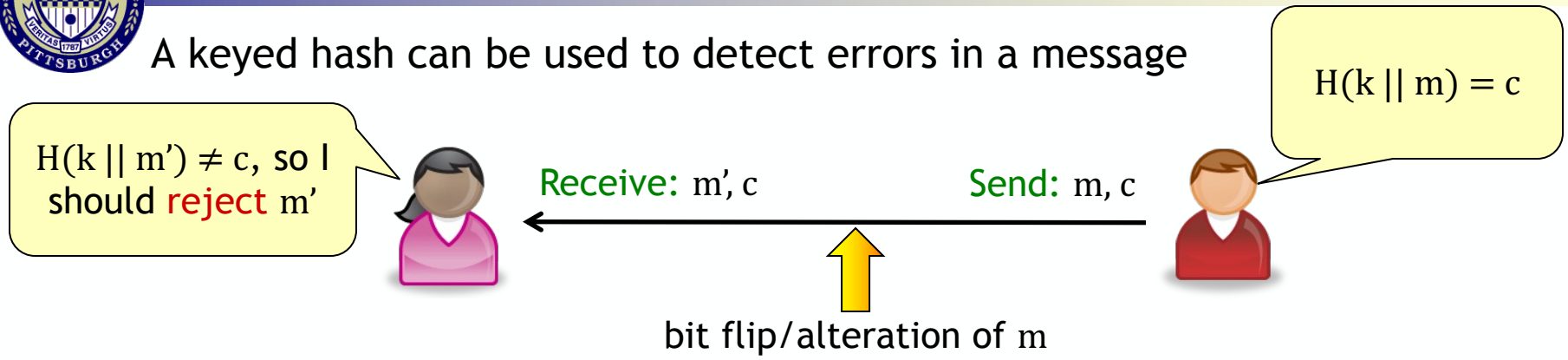
Why is a static IV ok?



Merkle and Damgård (separately) showed that the resulting hash function is secure if the compression function is collision resistant

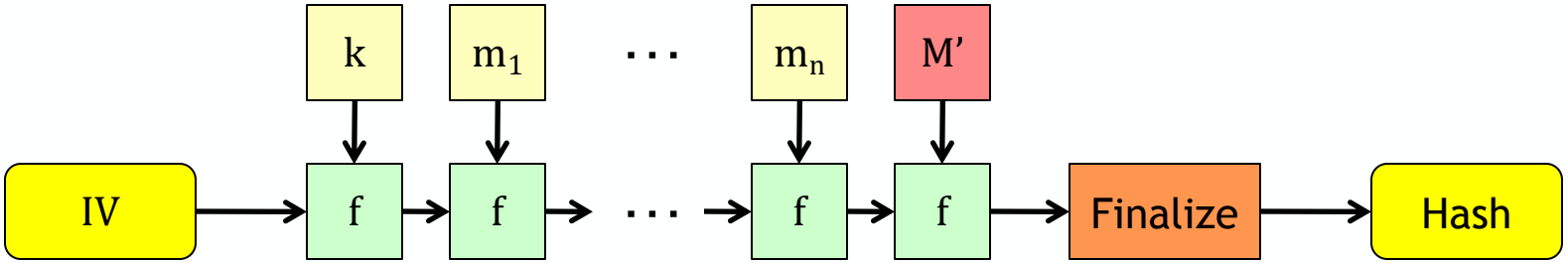
Although hashes are unkeyed functions, they can be used to generate MACs

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



Unfortunately, this isn't *totally* secure…

- It can be easy to append more data and update the MAC accordingly!

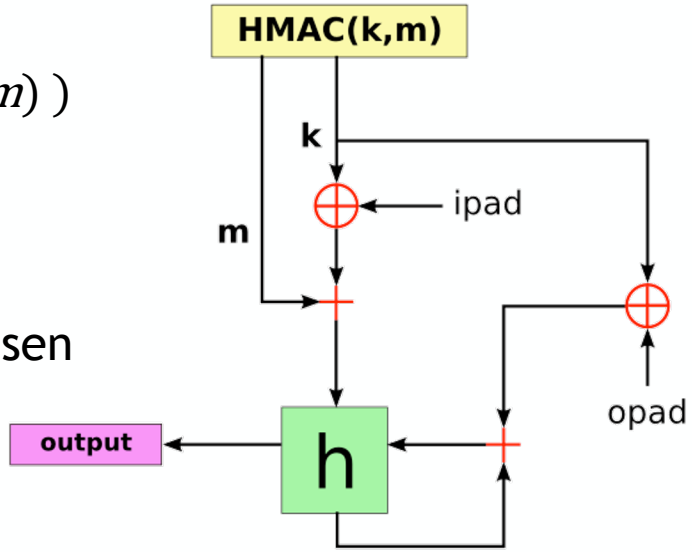


There are also attacks against $H(m \,||\, k)$ and $H(k \,||\, m \,||\, k)$!

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

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

- $\text{opad} = 01011010$
- $\text{ipad} = 00110110$



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

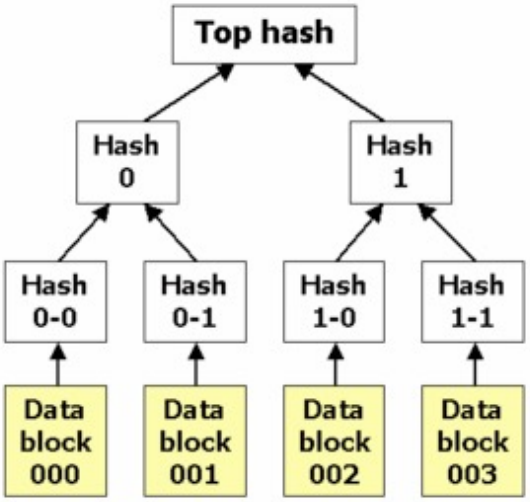
Note that *H* can be any hash function. For example, HMAC-SHA-1 is the name of the HMAC function built using the SHA-1 hash function.

Benefits of HMAC:

- Hash functions are faster than block ciphers
- Good security properties
- 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 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

Image from Wikipedia

Briefly, the history of hash functions

1989: MD2 published by Ron Rivest

- MD was never published
- Standardized in RFC 1319

1990: SNEFRU published by Ralph Merkle (Xerox)

- Much faster than MD2
- 1992: Broken* (Biham and Shamir)

1990: MD4 published in response (Rivest, RFC 1320)

- MD3 not published
- Even faster than SNEFRU
- 1992: Weaknesses found (den Boer and Bosselaers)

1991: MD5 (Rivest, RFC 1321)

- A bit slower than MD4, but fixed weaknesses

(MD4 and MD5 were also eventually broken)

Briefly, the history of hash functions

1993: NIST proposed SHA (NSA)

- Similar to MD5: a bit slower, improved strength
  - Output increased from 128 to 160 bits

1995: SHA-1 (NSA)

- Revision of SHA to fix discovered weaknesses
- SHA replaced and retroactively renamed to SHA-0

2004: SHA-2 (NSA)

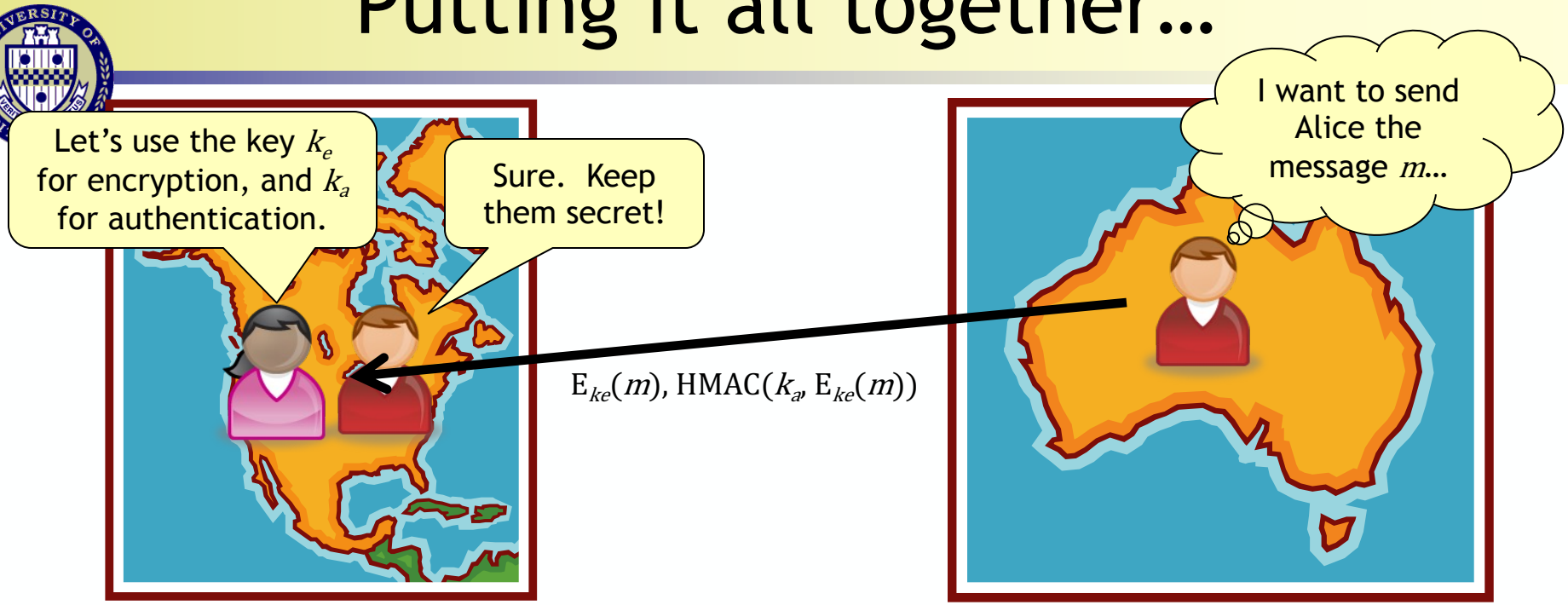
- Wanted longer hash length
- Variants with 224-, 256-, 384-, and 512-bit outputs
- Used alongside SHA-1

2005–2019: Various attacks published against SHA-1

- Initially, ways of finding collisions only with fewer rounds
- 2011: Collision in full SHA-1 in 2^{60} (~$2.77 M)
- 2015: Collision with non-standard IV (~$75–120 K)
- 2017: Two PDF files published with the same SHA-1

2006–2015: SHA-3 design competition (NIST)

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