Applied Cryptography
Week 5: Hash Functions and Keyed Hashing

Bernardo Portela

M:ERSI, M:SI - 24

What is a Hash Function?

Hash functions are everywhere

- Key derivation
- Digest for authentication
- Randomness extraction
- Password protection
- Proofs of work

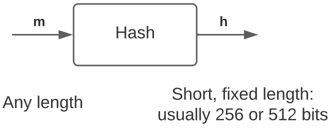
What is a Hash Function?

Hash functions are everywhere

- Key derivation
- Digest for authentication
- Randomness extraction
- Password protection
- Proofs of work

Not only in crypto:

- Indexing in version management
- Deduplication in cloud storage systems
- File integrity in intrusion detection



Describing Hash Functions

The hash output is short, aka hash, fingerprint or digest

Cryptographic hash functions give strong security guarantees

Use hash as an identifier

- Cryptographic hash functions cannot be injective

Describing Hash Functions

The hash output is short, aka hash, fingerprint or digest

Cryptographic hash functions give strong security guarantees

Use hash as an identifier

- Cryptographic hash functions cannot be injective
    - **Why?**

Describing Hash Functions

The hash output is short, aka hash, fingerprint or digest

Cryptographic hash functions give strong security guarantees

Use hash as an identifier

- Cryptographic hash functions cannot be injective
  - **Why?**
- Yet they should be *well distributed* and *unpredictable*
- Hash values can identify arbitrarily large inputs

Describing Hash Functions

The hash output is short, aka hash, fingerprint or digest

Cryptographic hash functions give strong security guarantees

Use hash as an identifier

- Cryptographic hash functions cannot be injective
  - **Why?**
- Yet they should be *well distributed* and *unpredictable*
- Hash values can identify arbitrarily large inputs

Signing $H(m)$ is **as secure** as signing m

Hash functions need to be deterministic and public

- Everyone should be able to recompute hash/identifier
- ... So what do we mean by security here?

Secure Cryptographic Hash Functions

Efficient algorithms with nice properties

- Unpredictable outputs
- Hard to find pre-images
- Hard to find collisions

Secure Cryptographic Hash Functions

Efficient algorithms with nice properties

- Unpredictable outputs
- Hard to find pre-images
- Hard to find collisions

Hash functions are validated heuristically

- Similar to process for AES
- International competition for select designs
- Competitors are scrutinized wrt security and performance
- Several rounds, so more eyes on small number of proposals
- Most recent one: SHA-3

#1: Pre-image resistance

It is *hard* to find the input that produced a given hash value

How can we establish this in concrete terms?

#1: Pre-image resistance

It is *hard* to find the input that produced a given hash value

How can we establish this in concrete terms?

Pre-image experiment

- Let $\mathcal{S}$ be the set of pre-images (domain)
- Let $\mathcal{R}$ be the set of images (range)
- Attacker is given a value $y \in \mathcal{R}$
- Attacker guesses $x \in \mathrm{S}$ and wins if $h(x) = y$

#2: Collision Resistance (CR)

- By definition, collisions **must exist**.
  - Recall that $|\mathcal{S}| >> |\mathcal{R}|$
- This can be argued from the *pidgeonhole principle*
  - If you have m holes and n pidgeons to put in these holes, if $n > m$, at least one hole will have more than one pidgeon!
- But can we find m_0 and m_1 s.t. $h(m_0) = h(m_1)$?

#2: Collision Resistance (CR)

- By definition, collisions **must exist**.
  - Recall that $|\mathcal{S}| >> |\mathcal{R}|$
- This can be argued from the *pidgeonhole principle*
  - If you have m holes and n pidgeons to put in these holes, if $n > m$, at least one hole will have more than one pidgeon!
- But can we find m_0 and m_1 s.t. $h(m_0) = h(m_1)$?

Suppose we have the *best possible hash function*?

#2: Collision Resistance (CR)

- By definition, collisions **must exist**.
  - Recall that $|\mathcal{S}| >> |\mathcal{R}|$
- This can be argued from the *pidgeonhole principle*
  - If you have m holes and n pidgeons to put in these holes, if $n > m$, at least one hole will have more than one pidgeon!
- But can we find m_0 and m_1 s.t. $h(m_0) = h(m_1)$?

Suppose we have the *best possible hash function*?
Q1: What could that be?

#2: Collision Resistance (CR)

- By definition, collisions **must exist**.
  - Recall that $|\mathcal{S}| >> |\mathcal{R}|$
- This can be argued from the *pidgeonhole principle*
  - If you have m holes and n pidgeons to put in these holes, if $n > m$, at least one hole will have more than one pidgeon!
- But can we find m_0 and m_1 s.t. $h(m_0) = h(m_1)$?

Suppose we have the *best possible hash function*?
Q1: What could that be?

- Lets think of the probability of collision
- Outputs are random, so $1/2^n$ where n is the output length
- Collision will be found if we check roughly 2^n pairs

Q2: Is CR harder or easier then pre-image resistance?

Breaking Hash Functions

Attack that finds a pre-image

- Search through all possible pre-images (brute-force)
- Consider a perfect hash function with output of n bits
- Cost: potentially more than 2^n operations!
- Absolutely unfeasible for modern hash functions
  - $n = 256$ for SHA-256 and BLAKE

Breaking Hash Functions

Attack that finds a pre-image

- Search through all possible pre-images (brute-force)
- Consider a perfect hash function with output of n bits
- Cost: potentially more than 2^n operations!
- Absolutely unfeasible for modern hash functions
    - $n = 256$ for SHA-256 and BLAKE

And if we want to find another pre-image?

- Nothing better than before
- Keep trying different values until you guess correctly

Breaking Hash Functions

Attack that finds a pre-image

- Search through all possible pre-images (brute-force)
- Consider a perfect hash function with output of n bits
- Cost: potentially more than 2^n operations!
- Absolutely unfeasible for modern hash functions
    - $n = 256$ for SHA-256 and BLAKE

And if we want to find another pre-image?

- Nothing better than before
- Keep trying different values until you guess correctly

But what if we only want to find a collision?

Finding Collisions

Collisions can be found with work $\sqrt{2^n}$, much better than 2^n!

Methodology

- Compute values like the brute-force attack
- Store them in a data structure *indexed by image value*
- Each new image value is searched in data structure
- Repeat until a collision is found

Finding Collisions

Collisions can be found with work $\sqrt{2^n}$, much better than 2^n!

Methodology

- Compute values like the brute-force attack
- Store them in a data structure *indexed by image value*
- Each new image value is searched in data structure
- Repeat until a collision is found

How many operations?

- After n values, we checked $n * (n-1)/2$ pairs **Q: why?**

Finding Collisions

Collisions can be found with work $\sqrt{2^n}$, much better than 2^n!

Methodology

- Compute values like the brute-force attack
- Store them in a data structure *indexed by image value*
- Each new image value is searched in data structure
- Repeat until a collision is found

How many operations?

- After n values, we checked $n * (n-1)/2$ pairs **Q: why?**
- Checking 2^n pairs takes roughly $\sqrt{2^n}$ values
- Overall complexity is that of finding the pre-image of a hash with $n/2$ bits of output (only half of the range)

The birthday paradox (not very paradoxical, just counterintuitive)

Implication of Birthday Attacks

For CR, hash outputs must be 2x security parameter

- 128-bit security $\rightarrow$ 256-bit hashes
- 256-bit security $\rightarrow$ 512-bit hashes

Implication of Birthday Attacks

For CR, hash outputs must be 2x security parameter

- 128-bit security $\rightarrow$ 256-bit hashes
- 256-bit security $\rightarrow$ 512-bit hashes

We can use security-parameter-sized hash outputs when:

- Security against arbitrary collisions is not required
- E.g. we might only need pre-image resistance
- Deriving a key from a secret input

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Never injective

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Never injective
- Easy to go from x to $f(x)$
- Hard to go from $f(x)$ to x

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Never injective
- Easy to go from x to $f(x)$
- Hard to go from $f(x)$ to x
- Pre-image resistance
  - Give me some x for which $f(x)$
- Collision resistance
  - Give me any x_1, x_2 for which $f(x_1) = f(x_2)$
  - Much easier!!

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Never injective
- Easy to go from x to $f(x)$
- Hard to go from $f(x)$ to x
- Pre-image resistance
  - Give me some x for which $f(x)$
- Collision resistance
  - Give me any x_1, x_2 for which $f(x_1) = f(x_2)$
  - Much easier!!
- For output of 2^n, collision can be found in $\approx 2^{\sqrt{n}}$
- So for 2^{128} resistance, output must be at least 2^{256}
- Birthday attack

Building Hash Functions

Two main approaches that use iterative processes

- **Merkle-Damgård construction:** Used for MD4, MD5, SHA-1, SHA-256, SHA-512. Relies on a $m + n$-to-n bits compression function to construct a hash function of output length n for arbitrary input lengths

Building Hash Functions

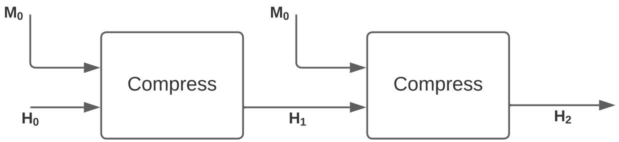
Two main approaches that use iterative processes

- **Merkle-Damgård construction:** Used for MD4, MD5, SHA-1, SHA-256, SHA-512. Relies on a $m + n$-to-n bits compression function to construct a hash function of output length n for arbitrary input lengths
- **Sponge construction:** Used for SHA-3, uses a l-bit permutation to construct a hash function for arbitrary input and output lengths

Merkle-Damgård Construction

All prominent hash functions from 80s-2000s.

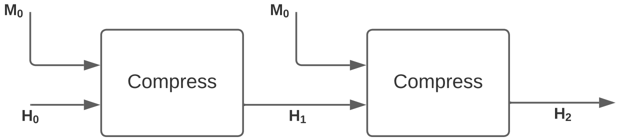
- H_0 is the initial value: constant and **public**
- M is broken into blocks of size m, $M_1, M_2, \ldots$



Merkle-Damgård Construction

All prominent hash functions from 80s-2000s.

- H_0 is the initial value: constant and **public**
- M is broken into blocks of size m, $M_1, M_2, \ldots$



- SHA-256: block size 512, output size 256 bits
- SHA-512: block size 1024, output size 512 bits
- What if messages are not of the same size as the block?

Merkle-Damgård Construction – Padding

Padding is always added to the message

- Append the message with a 1 bit
- Fill with zeros up to 64/128 bits away from the block end
- Last 64/128 bits encode the message length in bits

Merkle-Damgård Construction – Padding

Padding is always added to the message

- Append the message with a 1 bit
- Fill with zeros up to 64/128 bits away from the block end
- Last 64/128 bits encode the message length in bits

E.g. we want to hash the 8-bit string 10101010 using SHA-256

Message is: 10101010**1**00000(. . .)000001000

Merkle-Damgård Construction – Padding

Padding is always added to the message

- Append the message with a 1 bit
- Fill with zeros up to 64/128 bits away from the block end
- Last 64/128 bits encode the message length in bits

E.g. we want to hash the 8-bit string 10101010 using SHA-256

Message is: 10101010**1**00000(...)000001000

Q: Can't we just pad by adding 0s?

Merkle-Damgård Construction – Security

Useful result

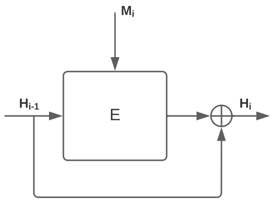
- Compression result is CR (for small inputs)
- Then the whole construction is CR (for arbitrary inputs)

To break the hash function you must break the compression function

So, does having a $2n$-to-n CR compression function solve all our problems?

Compression Functions: Davis-Meyer

All popular MD constructions use the Davis-Meyer construction:

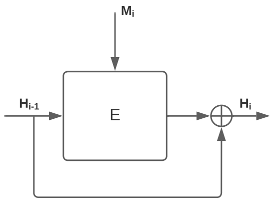


Block ciphers used as compression functions!

- Message is the encryption key!
- Construction creates a fixed point when $H_{i-1} = D(M_i, 0)$

Compression Functions: Davis-Meyer

All popular MD constructions use the Davis-Meyer construction:



Block ciphers used as compression functions!

- Message is the encryption key!
- Construction creates a fixed point when $H_{i-1} = D(M_i, 0)$

$$H_i = E(M_i, H_{i-1}) \oplus H_{i-1}$$
$$H_i = E(M_i, D(M_i, 0)) \oplus D(M_i, 0)$$
$$H_i = H_{i-1}$$

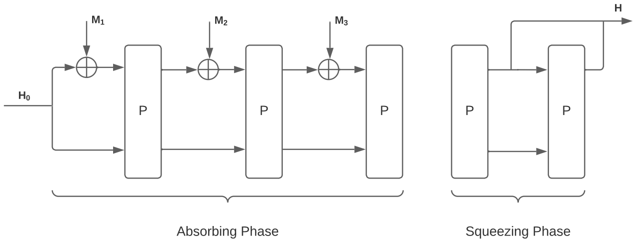
Sponge Construction

A more recent alternative to the MD is the sponge construction

It relies on a fixed (non-keyed) permutation

Very Versatile

- Varying input/output lengths
- PRGs and stream ciphers
- PRFs and keyed hashes



Sponge Construction – Description

Sponge operates in two phases: absorb and squeeze. The state is the same size w as the permutation input

Sponge Construction – Description

Sponge operates in two phases: absorb and squeeze. The state is the same size w as the permutation input

Absorb

- Fixed initial value h_0, gradually accumulate message into state
- Message broken in blocks of size r (rate)
- Block is smaller than state size
- Block XOR'ed into state
- Permutation recomputed

Sponge Construction – Description

Sponge operates in two phases: absorb and squeeze. The state is the same size w as the permutation input

Absorb

- Fixed initial value h_0, gradually accumulate message into state
- Message broken in blocks of size r (rate)
- Block is smaller than state size
- Block XOR'ed into state
- Permutation recomputed

Squeeze

- Dual process iteratively constructs output
- Output constructed block by block
- Permutation computed over the entire state
- Block-sized part of the state is accumulated in the output

MD5

- Broken! 128-bit output
- Most popular hash function until broken in 2005
- These days, it takes seconds to find collisions
- The SHA function family (next) uses a similar design

Secure Hash Function (SHA)

Standardized by NIST in the US. International *de facto* standard

SHA-0 published in 93', replaced with SHA-1 in 95'

- Both with 160-bit outputs
- Vulnerability not public at the time
- Later discovered collision attack in $2^{60} << 2^{80}$ operations
- More recent attacks reduced it to 2^{33}

Secure Hash Function (SHA)

Standardized by NIST in the US. International *de facto* standard

SHA-0 published in 93', replaced with SHA-1 in 95'

- Both with 160-bit outputs
- Vulnerability not public at the time
- Later discovered collision attack in $2^{60} << 2^{80}$ operations
- More recent attacks reduced it to 2^{33}

SHA-1 remained unbroken until quite recently – (**2017**)

Most applications currently use SHA-2 (256 or 512 bits)

- Same design principles; larger parameters

Future applications adopting SHA-3 evolve to the Sponge

- Flexible output size is very useful!

SHA-1 Internals

- Merkle-Damgård, with Davis-Meyer compression function
- Block cipher used in compression function called SHACAL
  - Block cipher with 160-bit block sizes!

SHA-1 Internals

- Merkle-Damgård, with Davis-Meyer compression function
- Block cipher used in compression function called SHACAL
    - Block cipher with 160-bit block sizes!
- Message blocks are 512-bits, hashes are 160-bits long
- Davis-Meyer addition (not XOR): five 32-bit additions
- Insecure! Expected collisions in 2^{63} ops in 2015, found in 2017

SHA-1 Internals

- Merkle-Damgård, with Davis-Meyer compression function
- Block cipher used in compression function called SHACAL
    - Block cipher with 160-bit block sizes!
- Message blocks are 512-bits, hashes are 160-bits long
- Davis-Meyer addition (not XOR): five 32-bit additions
- Insecure! Expected collisions in 2^{63} ops in 2015, found in 2017

```
SHA1-blockcipher(a, b, c, d, e, M) {
  W = expand(M);
  for i = 0 to 79 { // K are constants
    new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
    (a, b, c, d, e) = (new, a, b >>> 2, c, d)
  }
  return (a, b, c, d, e)
}
```

SHA-2 Family

- Family of 4 hash functions
  - SHA-224;256;384;512

SHA-2 Family

- Family of 4 hash functions
  - SHA-224;256;384;512
- Three digit identifier defines the output length
- Increased parameters and improved internal block ciphers
- SHA-224 and 256 still use 512 bit blocks (64 rounds)
  - SHA-224 is exactly the same as SHA-256, but has different IV and truncated output
  - SHA-384 and SHA-512 are similarly related
- SHA-512 compression function very similar, but has 80 rounds

SHA-2 Family

- Family of 4 hash functions
  - SHA-224;256;384;512
- Three digit identifier defines the output length
- Increased parameters and improved internal block ciphers
- SHA-224 and 256 still use 512 bit blocks (64 rounds)
  - SHA-224 is exactly the same as SHA-256, but has different IV and truncated output
  - SHA-384 and SHA-512 are similarly related
- SHA-512 compression function very similar, but has 80 rounds

No non-generic attacks exist on these hash functions

- Still SHA-3 was (prudently) developed with different design
- Also has the benefit of varying sized outputs
- Good to generate keys!

SHA-3

- Keccack selected in 2009
- 3-year NIST SHA-3 competition
- Competition called for new design, if SHA-2 gets attacked

SHA-3

- Keccack selected in 2009
- 3-year NIST SHA-3 competition
- Competition called for new design, if SHA-2 gets attacked

Keccack is very different and very flexible

- Sponge based with 1600-bits permutation (in SHA-3)
- Blocks can be 1152, 1088, 832 or 576 bits
- Corresponding to 224, 256, 384 or 512 bit outputs
- As a bonus we get the SHAKE functions
  - SHAKE128 and SHAKE256
  - eXtendable Output Functions (XOFs)
  - You can specify output length

Key Takeaways

- Two main constructions: MD and Sponge

Key Takeaways

- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2
  - Sequential compression of message – DM
  - Davis-Meyer compresses using a block cipher

Key Takeaways

- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2
  - Sequential compression of message – DM
  - Davis-Meyer compresses using a block cipher
- Sponge
  - Used in SHA3 and SHAKE
  - Absorb phase permutes message
  - Squeeze phase retrieves the output

Key Takeaways

- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2
  - Sequential compression of message – DM
  - Davis-Meyer compresses using a block cipher
- Sponge
  - Used in SHA3 and SHAKE
  - Absorb phase permutes message
  - Squeeze phase retrieves the output
- SHA-2 and SHA-3 currently the *de facto* standards

MACs as Keyed Hashes

Short Summaries of Potentially Large Messages

- Called a hash if everything is public
- Keyed hashes allows for conditional hash computation

MACs as Keyed Hashes

Short Summaries of Potentially Large Messages

- Called a hash if everything is public
- Keyed hashes allows for conditional hash computation

Message Authentication Codes – MACs

- Symmetric Authentication $t \leftarrow \mathsf{MAC}(k, m)$
- t guarantees that m was produced by someone that knows k
- Implies message m was not changed since its creation
- Digital signatures in the symmetric paradigm!

Message Authentication Codes

Typical use of MACs – SSH, IPSec, TLS

- Two parties was message authentication and integrity
- Some form of set-up/agreement to establish common key k
- Sender computes $t \leftarrow \mathrm{MAC}(k, m)$ and sends (m, t)
- Receiver gets (m, t), recomputes $t' \leftarrow \mathrm{MAC}(k, m)$
- If $t \neq t'$, message is rejected!

Message Authentication Codes

Typical use of MACs – SSH, IPSec, TLS

- Two parties was message authentication and integrity
- Some form of set-up/agreement to establish common key k
- Sender computes $t \leftarrow \mathrm{MAC}(k, m)$ and sends (m, t)
- Receiver gets (m, t), recomputes $t' \leftarrow \mathrm{MAC}(k, m)$
- If $t \neq t'$, message is rejected!

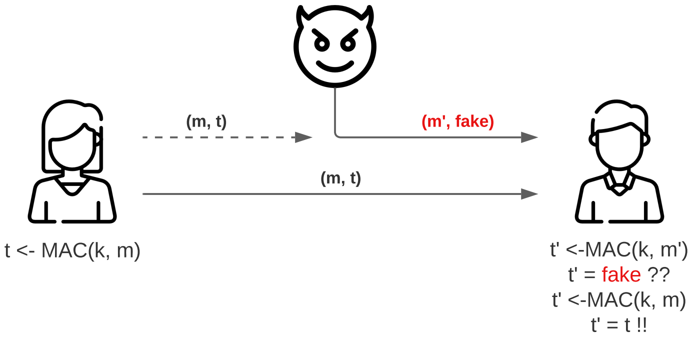
Acceptance means m was produced while knowing k

In this process, message is public!

MACs do not give <u>confidentiality</u>. They provide <u>integrity</u>

Its orthogonal to encryption. In real-world applications, we will need to combine these

Authentication and Message Integrity



- No possibility of computing t without k implies
- Adversary cannot change the message
- Adversary cannot conjure new messages

MAC Security

Standard notion is UF-CMA

- Goal: Unforgeability
- Adversary power: Chosen Message Attacks

MAC Security

Standard notion is UF-CMA

- Goal: Unforgeability
- Adversary power: Chosen Message Attacks

Security Experiment

- Experiment generates a key k
- Adversary (adaptively) sends m to get $t \leftarrow \text{MAC}(k, m)$
- Eventually, attacker outputs (m^*, t^*)

Attacker wins if $t^* = \text{MAC}(k, m^*)$, and if t^* was not produced by the experiment. Contrary to IND-CPA, a victory here implies a broken MAC scheme.

MAC Security Nuances

- MAC on its own does not protect against replay attacks
- Suppose a network scenario
    - Attacker sees authenticated message (m, t)
    - Delivers (m, t) multiple times
    - MAC will verify every time!

MAC Security Nuances

- MAC on its own does not protect against replay attacks
- Suppose a network scenario
    - Attacker sees authenticated message (m, t)
    - Delivers (m, t) multiple times
    - MAC will verify every time!
- Simple technique: impose message never repeats in network
- Sequence numbers
    - Prepend counter and keep counter as state in both sides
    - Prepend timestamp (local clock reading)
    - How should the receiver operate in both cases?

Some Context

MACs constructed from hash functions and block ciphers

Simplest construction: prefix key

$$\mathrm{MAC}(k, m) = \mathrm{H}(K||M) \text{ or } \mathrm{PRF}(k, m) = \mathrm{H}(K||M)$$

Some Context

MACs constructed from hash functions and block ciphers

Simplest construction: prefix key

$$\mathrm{MAC}(k, m) = \mathrm{H}(K||M) \text{ or } \mathrm{PRF}(k, m) = \mathrm{H}(K||M)$$

MD yields insecure MAC and PRF!

- Given (m, t), attacker outputs $\mathrm{H}(K||M||pad||M')$
- This can be computed just from t' and m'
- **Length extension attack**

Some Context

MACs constructed from hash functions and block ciphers

Simplest construction: prefix key

$$\mathrm{MAC}(k, m) = \mathrm{H}(K||M) \text{ or } \mathrm{PRF}(k, m) = \mathrm{H}(K||M)$$

MD yields insecure MAC and PRF!

- Given (m, t), attacker outputs $\mathrm{H}(K||M||pad||M')$
- This can be computed just from t' and m'
- **Length extension attack**

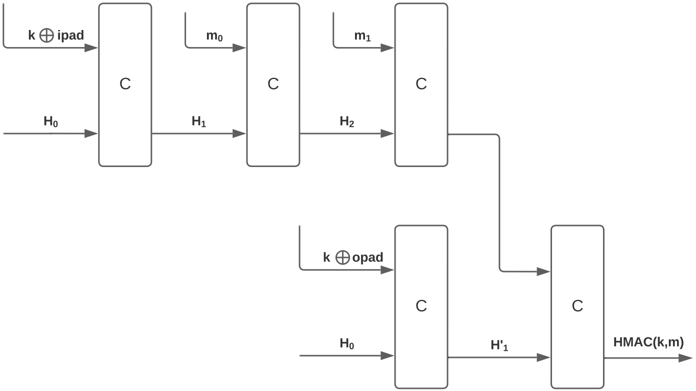
A consideration in SHA-3 construction

- Abandon MD construction
- Include explicit keyed hash

HMAC Construction

When instantiated with MD construction

- Compression function is PRF → Secure MAC
- HMAC is simply $\mathrm{H}((K \oplus opad)||\mathrm{H}((k \oplus ipad)||m))$
- *ipad* and *opad* are constraints: align to block size



On Collision-based Forgeries

Hash function collisions $\rightarrow$ hash-based MAC forgeries

However, attacker cannot easily search for them w/o key

On Collision-based Forgeries

Hash function collisions $\rightarrow$ hash-based MAC forgeries

However, attacker cannot easily search for them w/o key

Collisions in MAC also yield forgeries

- True for **any** MAC
- Collision occur when $\sqrt{2^n}$ MACs are issued

Building MACs from Block Ciphers

We have seen block ciphers $\rightarrow$ hash functions $\rightarrow$ MACs

But there are also direct constructions: block ciphers $\rightarrow$ MACs

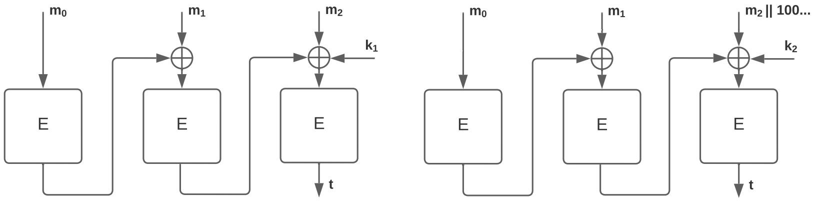
CMAC

- Used in IPSec
- CMAC improves on CBC-MAC (which was broken!)
- Use CBC mode of operation
- Fix IV to all zero blocks
- Take the last ciphertext block as a tag

CMAC Internals

CMAC fixes CBC-MAC by processing last block differently

- All blocks except last are processed like CBC-MAC
- Keys k_1 and k_2 derived from k
  - $l \leftarrow E(k, 0)$
  - $k_1 = (l << 1) \oplus (0x00..0087 * LSB(l))$
  - $k_2 = (k_1 << 1) \oplus (0x00..0087 * LSB(k_1))$



Custom MAC Constructions

More efficient MAC constructions are designed from scratch

Poly1305 is one such construction by D. J. Bernstein

Based on

- Universal Hash Functions
- Wegman-Carter construction

Universal Hash Functions

UHF are a Weak form of Hashing

- Don't need to be collision resistance
- Parametrised by a key $\mathrm{UH}(k, m)$
- Guarantee that, for two fixed messages $m_0 \neq m_1$:

$$\Pr[\mathrm{UH}(k, m_0) = \mathrm{UH}(k, m_1)] \leq \epsilon$$

- Considering random k and very small ϵ

Universal Hash Functions

UHF are a Weak form of Hashing

- Don't need to be collision resistance
- Parametrised by a key $\mathrm{UH}(k, m)$
- Guarantee that, for two fixed messages $m_0 \neq m_1$:

$$\Pr[\mathrm{UH}(k, m_0) = \mathrm{UH}(k, m_1)] \leq \epsilon$$

- Considering random k and very small ϵ

No other security requirements $\rightarrow$ easy to construct

We can use a universal hash function as a MAC

Provided that we only authenticate **one message!**

Wegman-Carter Construction

How to circumvent this limitation?

- Use a PRF to strengthen the UH
- Converts a UH into a fully secure MAC
- AES can fill the PRF role!

Wegman-Carter Construction

How to circumvent this limitation?

- Use a PRF to strengthen the UH
- Converts a UH into a fully secure MAC
- AES can fill the PRF role!

Intuition: Encrypt Universal Hash Value

$$\mathrm{UH}(k_1, m) \oplus \mathrm{PRF}(k_2, n)$$

- The full MAC key is (k_1, k_2)
- n is a public value that must never repeat
    - A.k.a. a nonce
- This can be kept as a counter, or generated at random

Poly1305-AES: Wegman-Carter in Practice

- Initial proposal used AES as the Wegman-Carter PRF
- The universal hash function uses prime $p^{130} - 5$

$$\text{Poly1305}((k_1, k_2), m) = (m_1 k + \ldots + m_n k^n \pmod{p}) + \text{AES}(k_2, n)$$

- Blocks are 128 bits and last block is padded with 100
- All blocks set bit 129, so MSB is 1
- The final addition is performed modulo 2^{128}
- TLS recommends Poly1305 with ChaCha20, rather than AES

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function f, one cannot produce $h(k, x)$ w/o k
- Provides <u>integrity</u>; ciphers give <u>confidentiality</u>

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function f, one cannot produce $h(k, x)$ w/o k
- Provides integrity; ciphers give confidentiality
- Just add a key as prefix!
    - Problem! Length extension attacks

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function f, one cannot produce $h(k, x)$ w/o k
- Provides integrity; ciphers give confidentiality
- Just add a key as prefix!
  - Problem! Length extension attacks
- HMAC
  - Input padding and output padding, both using the key

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function f, one cannot produce $h(k, x)$ w/o k
- Provides <u>integrity</u>; ciphers give <u>confidentiality</u>
- Just add a key as prefix!
  - Problem! Length extension attacks
- HMAC
  - Input padding and output padding, both using the key
- CMAC
  - Do AES-CBC without IV; return the last block
  - With a twist to prevent prefix extension

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function f, one cannot produce $h(k, x)$ w/o k
- Provides integrity; ciphers give confidentiality
- Just add a key as prefix!
  - Problem! Length extension attacks
- HMAC
  - Input padding and output padding, both using the key
- CMAC
  - Do AES-CBC without IV; return the last block
  - With a twist to prevent prefix extension
- Wegman-Carter
  - Use a UHF for a unique message
  - XOR it with an encryption of a nonce
  - Used in AES-GCM (next class)

Applied Cryptography
Week 5: Hash Functions and Keyed Hashing

Bernardo Portela

M:ERSI, M:SI - 24