Computer Security Foundations
Week 9: MACs and Authenticated Encryption

Bernardo Portela

L.EIC - 24

What is a Hash Function?

Hash functions are everywhere

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

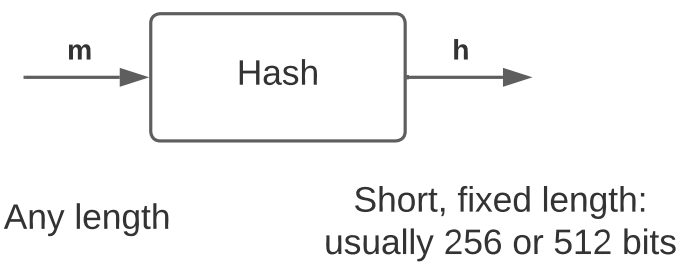
What is a Hash Function?

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



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

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)

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

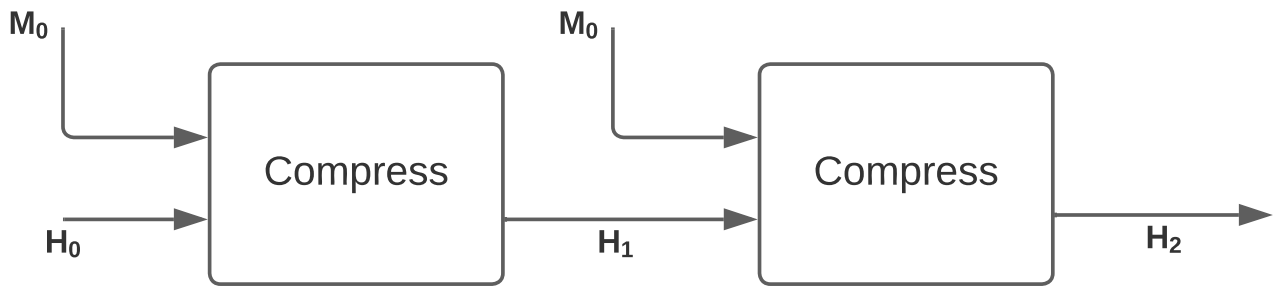
Two main approaches that use iterative processes

- **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.

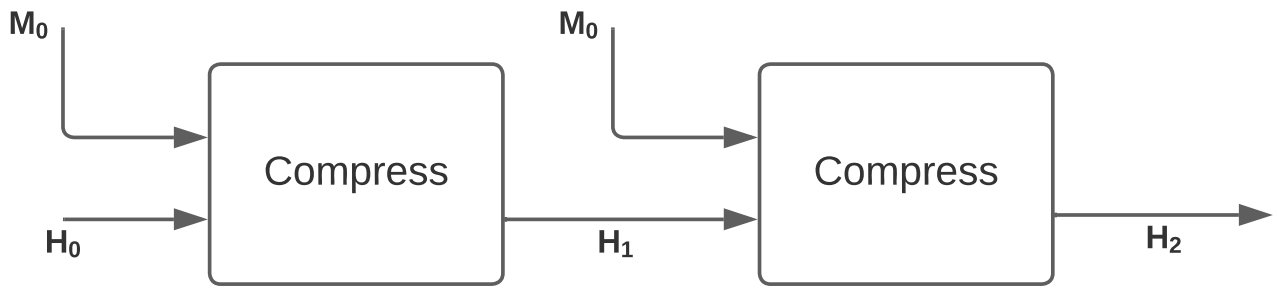
- H_0 is the initial value: constant and **public**
- M is broken into blocks of size m, $M_1, M_2, \ldots$



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?

Sponge Construction

Absorb

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

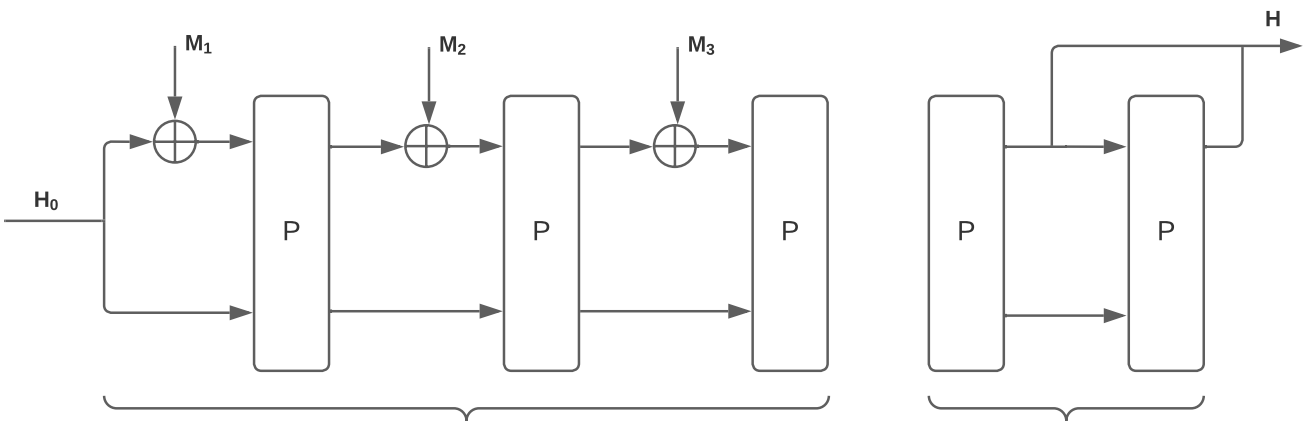
Sponge Construction

Absorb

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

Squeeze

- Dual process iteratively constructs output
- Output constructed block by block



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

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - 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
  - Easy to go from x to $f(x)$
  - Hard to go from $f(x)$ to x
- For output of 2^n, collision can be found in $\approx \sqrt{2^n} = 2^{\frac{n}{2}}$
  - So for 2^{128} resistance, output must be at least 2^{256}

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Easy to go from x to $f(x)$
  - Hard to go from $f(x)$ to x
- For output of 2^n, collision can be found in $\approx \sqrt{2^n} = 2^{\frac{n}{2}}$
  - So for 2^{128} resistance, output must be at least 2^{256}
- Two main constructions: MD and Sponge

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Easy to go from x to $f(x)$
  - Hard to go from $f(x)$ to x
- For output of 2^n, collision can be found in $\approx \sqrt{2^n} = 2^{\frac{n}{2}}$
  - So for 2^{128} resistance, output must be at least 2^{256}
- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Easy to go from x to $f(x)$
  - Hard to go from $f(x)$ to x
- For output of 2^n, collision can be found in $\approx \sqrt{2^n} = 2^{\frac{n}{2}}$
  - So for 2^{128} resistance, output must be at least 2^{256}
- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2
- Sponge
  - Used in SHA3 and SHAKE

Key Takeaways

- Hash functions are one-way functions
  - From any sized inputs to fixed-size output
  - Easy to go from x to $f(x)$
  - Hard to go from $f(x)$ to x
- For output of 2^n, collision can be found in $\approx \sqrt{2^n} = 2^{\frac{n}{2}}$
  - So for 2^{128} resistance, output must be at least 2^{256}
- Two main constructions: MD and Sponge
- Merkle-Damgård
  - Used in MD5; SHA1; SHA2
- Sponge
  - Used in SHA3 and SHAKE
- 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 $t \leftarrow \mathrm{MAC}(k, m)$
- 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!

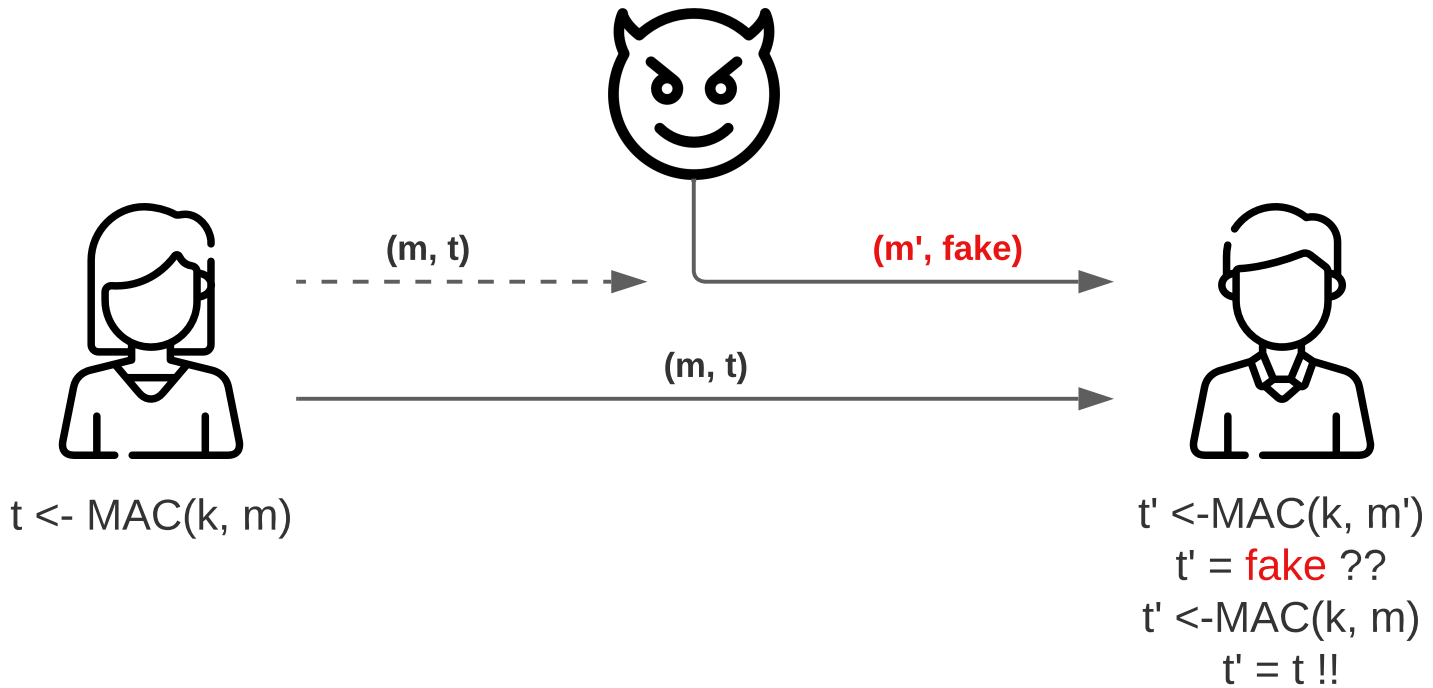
Acceptance means m was produced while knowing k

In this process, message is public!

MACs do not give confidentiality. They provide integrity

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

Some Context

MACs constructed from hash functions and block ciphers

Simplest construction: prefix key

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

Some Context

MACs constructed from hash functions and block ciphers

Simplest construction: prefix key

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

MD yields insecure MAC!

- 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

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

MD yields insecure MAC!

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

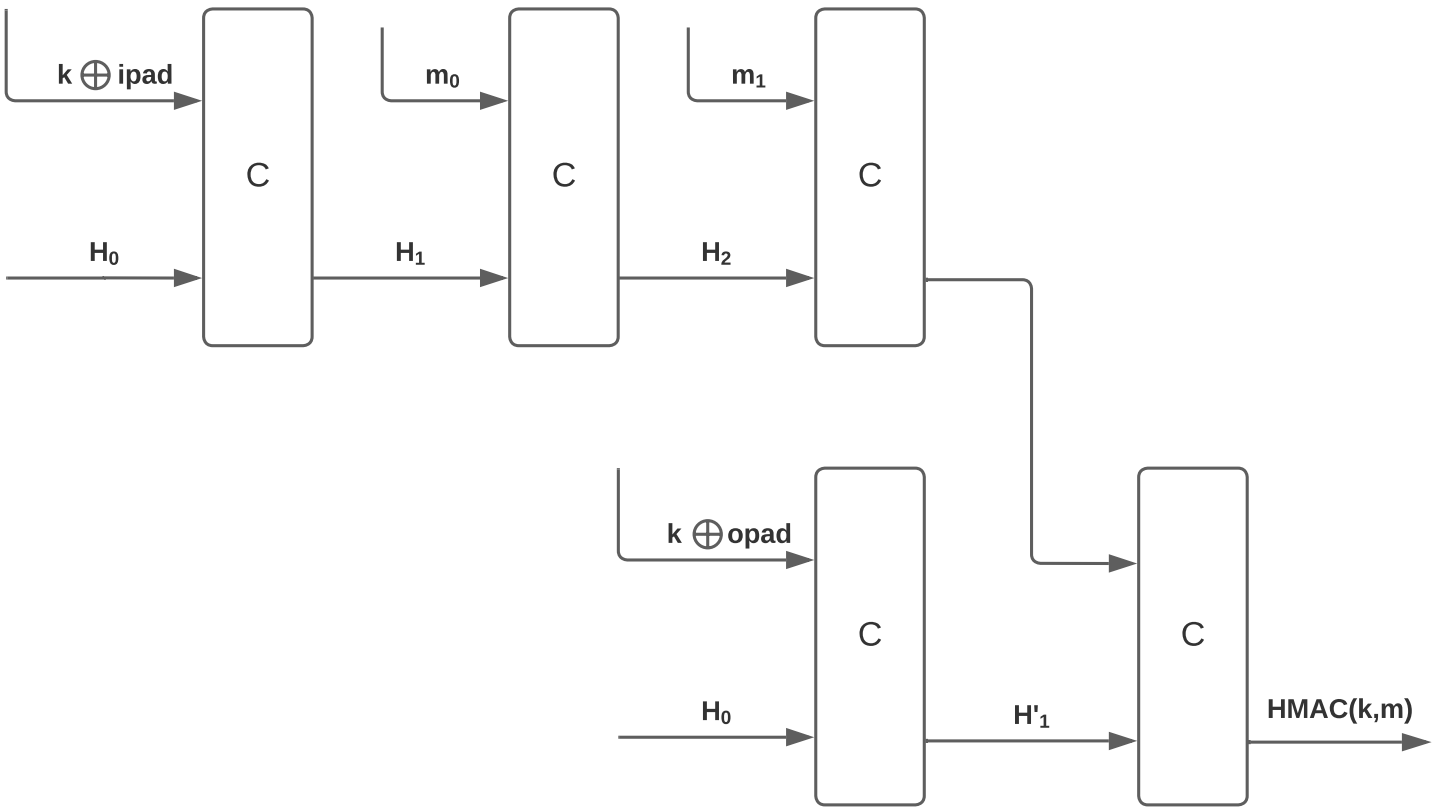
A consideration in SHA-3 construction

- Abandon MD construction
- Include explicit keyed hash

HMAC Construction

When instantiated with MD construction

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



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

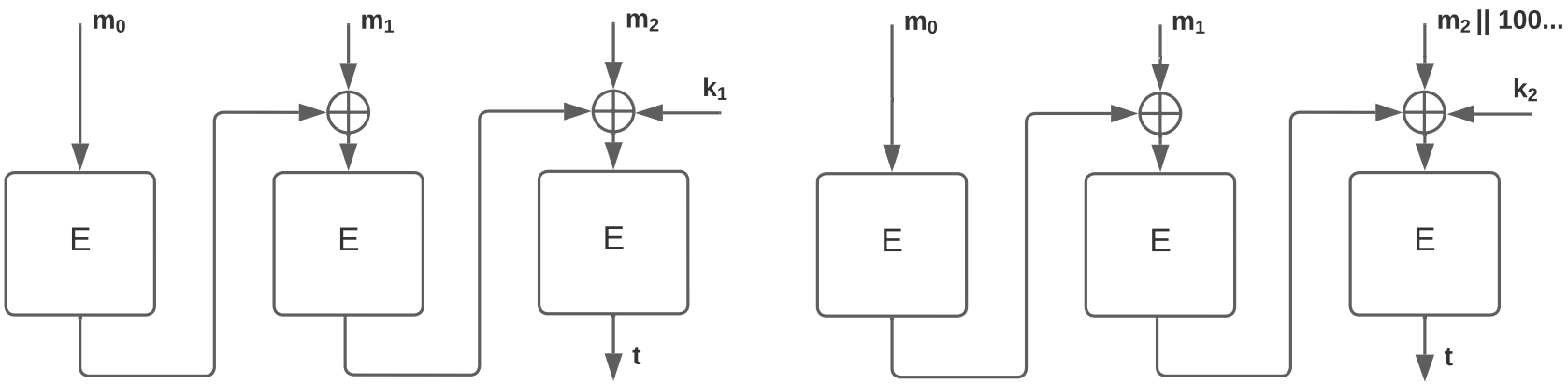
CMAC

- 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
  - $l \leftarrow E(k, 0)$
  - $k_1 = (l << 1) \oplus (0x00..0087 * LSB(l)))$
  - $k_2 = (k_1 << 1) \oplus (0x00..0087 * LSB(k_1)))$



Universal Hash Functions

UHF are a Weak form of Hashing

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

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

- 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 $\mathsf{UH}(k, m)$
- Guarantee that, for two fixed messages $m_0 \neq m_1$:

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

- 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 $2^{130} - 5$

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

- Blocks are 128 bits and last block is padded with 100
- 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 h, one cannot produce $h(k, x)$ w/o k
- Provides integrity; ciphers give confidentiality

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function h, 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 h, 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 h, 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

Key Takeaways

- Keyed hashing allows for message authentication
- For hash function h, 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!)

Why Authenticated Encryption?

Any secure channel in practice uses authenticated encryption

- Messages need to be confidential
- Messages need to be authentic
- Messages should not be repeated/omitted/removed

Why Authenticated Encryption?

Any secure channel in practice uses authenticated encryption

- Messages need to be confidential
- Messages need to be authentic
- Messages should not be repeated/omitted/removed

Encryption provides confidentiality

MACs provide authenticity

Why Authenticated Encryption?

Any secure channel in practice uses authenticated encryption

- Messages need to be confidential
- Messages need to be authentic
- Messages should not be repeated/omitted/removed

Encryption provides confidentiality

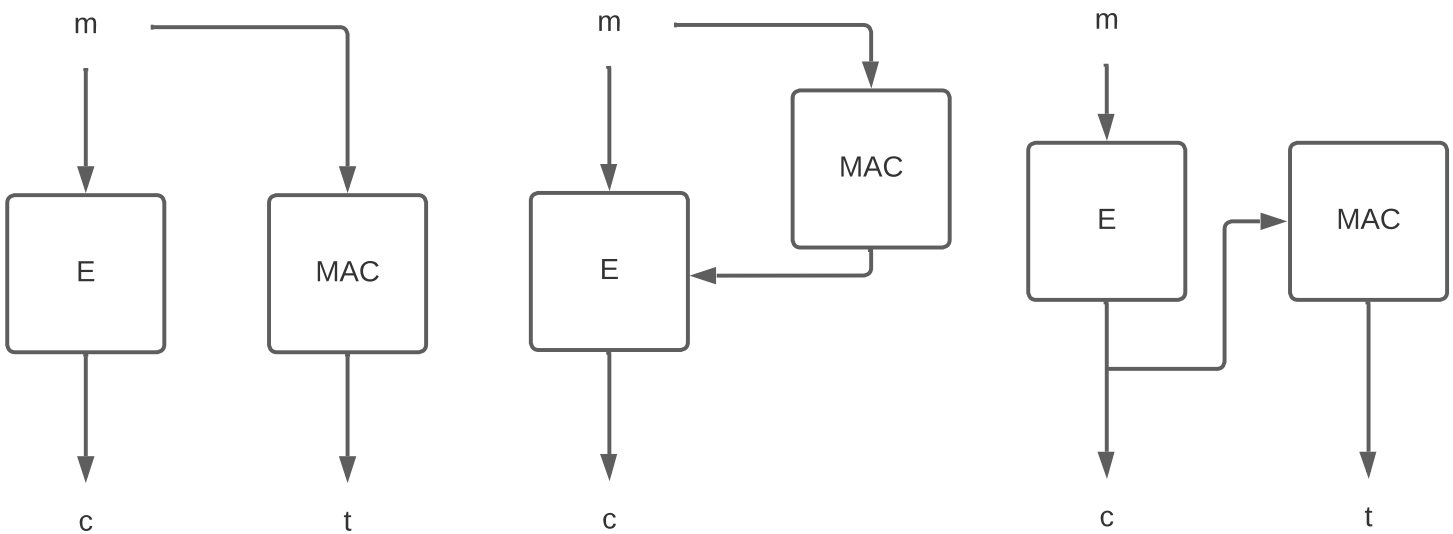
MACs provide authenticity

Authenticated public meta-information (e.g. sequence numbers) is used to solve the third point – Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption using MACs

The good, the bad and the ugly

- Encrypt-and-MAC: encryption and MAC of the message
- MAC-then-Encrypt: Encrypt message and its authentication
- Encrypt-then-MAC: Authenticate the message encryption



Encrypt-and-MAC MAC-then-encrypt Encrypt-then-MAC

Encrypt-and-MAC

AE with $k = (k_1, k_2)$ done with parallel processing

- $c \xleftarrow{\$} E(k_1, m)$
- $t \leftarrow MAC(k_2, m)$
- Output: (c, t)

Encrypt-and-MAC

AE with $k = (k_1, k_2)$ done with parallel processing

- $c \xleftarrow{\$} E(k_1, m)$
- $t \leftarrow MAC(k_2, m)$
- Output: (c, t)

Problems – The bad

- Potentially malicious c decrypted before authentication
- MACs not designed to ensure confidentiality!
- Construction can be secure for *some* MACs...
  - Very easy to make mistakes
- Used in SSH: $MAC(k_2, m||n)$, where n is the sequence number

MAC-then-Encrypt

AE with $k = (k_1, k_2)$ done sequentially

- $t \leftarrow MAC(k_1, m)$
- $c \leftarrow\!\!\!\!\!\!\$\ \ E(k_2, m||t)$
- Output: c

MAC-then-Encrypt

AE with $k = (k_1, k_2)$ done sequentially

- $t \leftarrow MAC(k_1, m)$
- $c \leftarrow\!\!\!\!\!\!\$\ E(k_2, m||t)$
- Output: c

Problems – The ugly

- Potentially malicious c decrypted before authentication
- Used in TLS until version 1.3
- Painful story with *padding oracle attacks*
  - Issue arises from the decryption before authentication
  - Did the decryption fail because of the padding, or because of the MAC?
  - Theoretical attack found – disregarded at first
  - Practical attack found a couple of years later: Lucky 13

Encrypt-then-MAC

AE with $k = (k_1, k_2)$ done sequentially

- $c \xleftarrow{\$} E(k_1, m)$
- $t \leftarrow MAC(k_2, c)$
- Output: c

Encrypt-then-MAC

AE with $k = (k_1, k_2)$ done sequentially

- $c \leftarrow\!\!\$\ E(k_1, m)$
- $t \leftarrow MAC(k_2, c)$
- Output: c

Advantages – The good

- Ciphertext not decrypted **unless it is authenticated**
- Useful against DoS attacks
  - MAC verification typically very fast
- Preferred method, except in legacy systems

Authenticated Encryption from Scratch

Modern AEADs are not black-box compositions of Enc+MAC.
Encryption/authentication layers visible in all constructions

Authenticated Encryption from Scratch

Modern AEADs are not black-box compositions of Enc+MAC.
Encryption/authentication layers visible in all constructions

Optimized for Performance

- Encryption/decryption of blocks in parallel
- Throughput, streamability, memory requirements, input fixing:
  - Implementation cost, e.g., need block cipher inverse?
  - Can start transmitting before encryption is complete?
  - Can start decryption before ciphertext is fully received?
  - Can discard plaintext/ciphertext block immediately (online)?
  - Can metadata be given at any point? Or must be fixed initially?

Authenticated Encryption from Scratch

Modern AEADs are not black-box compositions of Enc+MAC.
Encryption/authentication layers visible in all constructions

Optimized for Performance

- Encryption/decryption of blocks in parallel
- Throughput, streamability, memory requirements, input fixing:
  - Implementation cost, e.g., need block cipher inverse?
  - Can start transmitting before encryption is complete?
  - Can start decryption before ciphertext is fully received?
  - Can discard plaintext/ciphertext block immediately (online)?
  - Can metadata be given at any point? Or must be fixed initially?

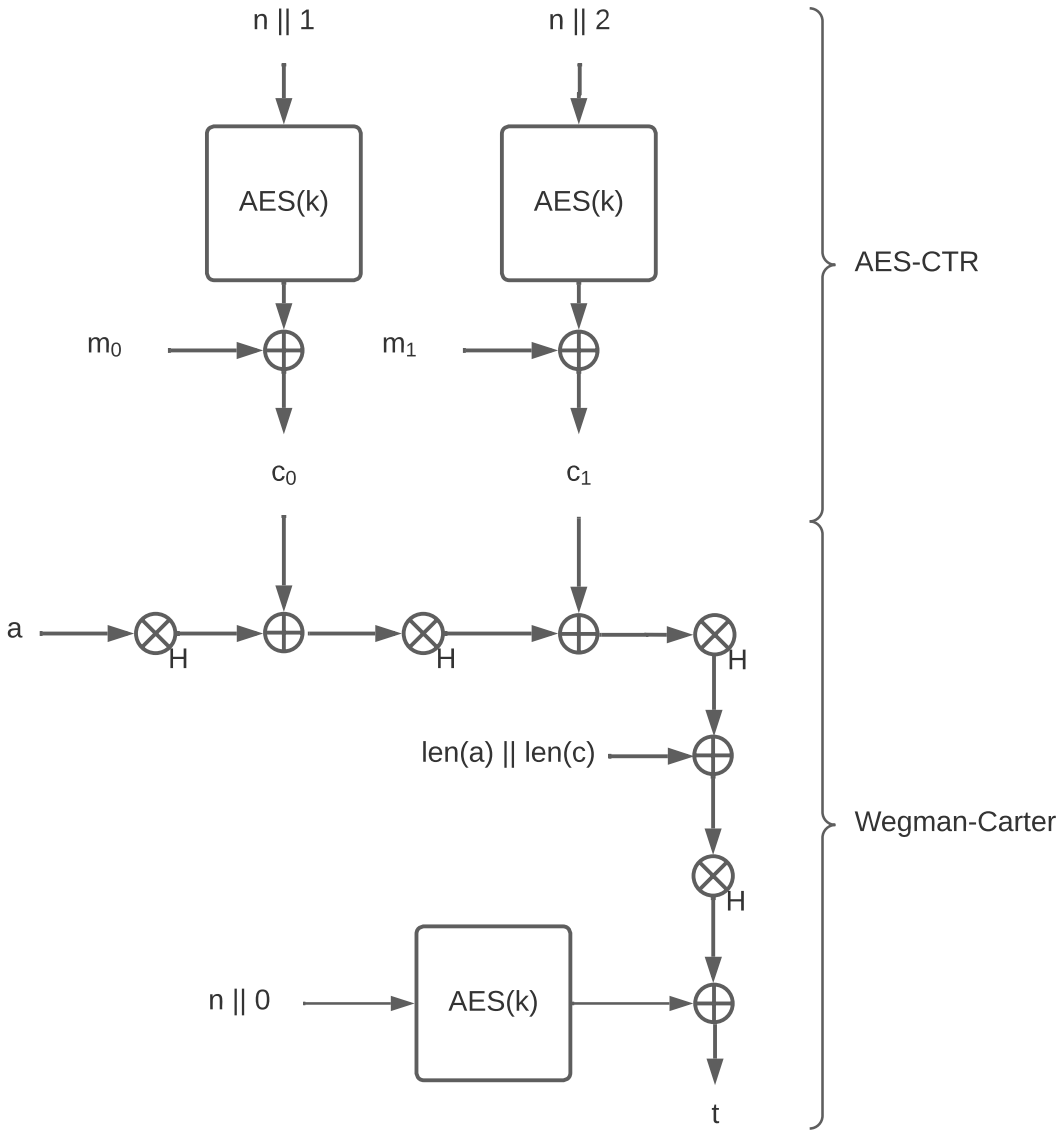
Streamability (a.k.a. online) $\rightarrow$ low memory requirements

- Very important for routers, TLS/https termination, etc.

Galois-Counter Mode

Most widely used AEAD:
IPSec, SSH, TLS

- GCM:
  - CTR plus auth layer
- Keys: 128 bits
- Nonce: 96 bits
- CTR starts at 1



Galois-Counter Mode: Authentication

Q: What type of AE is this?

Galois-Counter Mode: Authentication

Q: What type of AE is this? Encrypt-then-MAC!

MAC uses Wegman-Carter construction

- Universal Hash function is called GHASH
- AES used as PRF to hash the value on input $n \,||\, 0$

Galois-Counter Mode: Authentication

Q: What type of AE is this? Encrypt-then-MAC!

MAC uses Wegman-Carter construction

- Universal Hash function is called GHASH
- AES used as PRF to hash the value on input $n \,||\, 0$

$GHASH(hk, c_1, \ldots, c_n)$ defined as follows

- $hk \leftarrow AES(k, 0)$
- Evaluate $P(x)$ defined by $(\text{pad0}(A), \text{pad0}(c), |A| \,||\, |C|)$ at hk
- Horner's formula: $P(x) = x * (x * (x * (\ldots) + a_2) + a_1) + a_0$
- Algebraic magic:
  - Super efficient in hardware (special processor instructions)

Efficiency of Galois Counter Mode

Encryption layer inherits parallelism from CTR mode

Authentication layer blocks if a is known only in the end

Efficiency of Galois Counter Mode

Encryption layer inherits parallelism from CTR mode

Authentication layer blocks if a is known only in the end

a known from the start $\rightarrow$ GCM streamable

- Ciphertext blocks computed and authenticated on the fly
- Authentication of previous block is accumulated while current block is being encrypted

Efficiency of Galois Counter Mode

Encryption layer inherits parallelism from CTR mode

Authentication layer blocks if a is known only in the end

a known from the start $\rightarrow$ GCM streamable

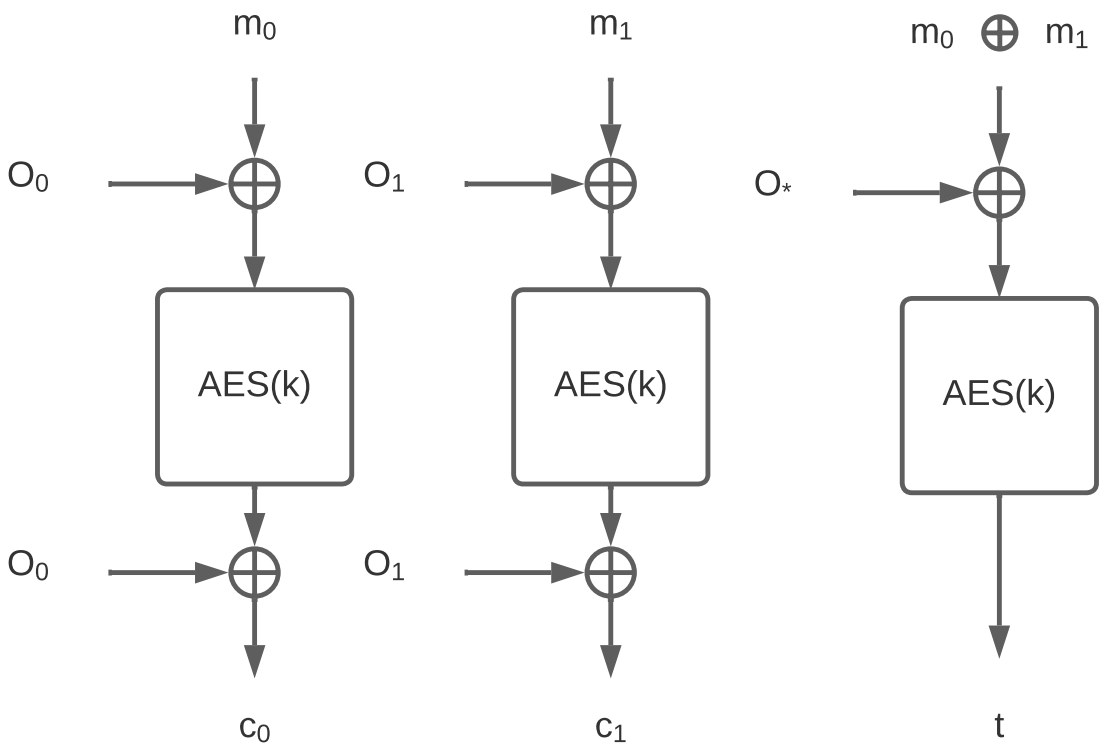
- Ciphertext blocks computed and authenticated on the fly
- Authentication of previous block is accumulated while current block is being encrypted

Could be faster...

- Authentication usually not computed fully in parallel
- HW implementations of auth layer slower than AES-CTR

(ChaCha20-Poly1305 not covered, but has similar structure)

Offset Codebook (OCB)



- Offset depends on the key and the nonce
- Incremented for each new block
- Last offset computed from last plaintext block processed

Using OCB as AEAD

- Very secure and efficient
- Implementations required licensing
- Patent-renewal fees intentionally not payed (see here)

Using OCB as AEAD

- Very secure and efficient
- Implementations required licensing
- Patent-renewal fees intentionally not payed (see here)

Similar to AES-GCM, OBC allows authenticated data $a_0, a_1, \ldots, a_n$

Tag computed as

$$T = E(k, S \oplus O_*) \oplus E(k, a_0 \oplus O_0) \oplus E(k, a_1 \oplus O_1) \oplus \ldots$$

where $S = m_0 \oplus m_1 \oplus \ldots$

Offset values for AD different from those used to encrypt $m_0, m_1, \ldots, m_n$

Security and Efficiency of OCB

Nonce reuse

- On nonce reuse, the attacker can identify block duplicates
  - E.g. block 3 of message 1 is similar to block 5 of message 2
  - GCM allows detection of multiple blocks, but also XOR differences for blocks in the same position
- Repeated nonces can break the authenticity of OCB
  - An attacker can combine blocks from messages authenticated with OCB to create another authenticated message
  - ...but unlike GCM, it cannot extract the underlying key!!

Security and Efficiency of OCB

Nonce reuse

- On nonce reuse, the attacker can identify block duplicates
  - E.g. block 3 of message 1 is similar to block 5 of message 2
  - GCM allows detection of multiple blocks, but also XOR differences for blocks in the same position
- Repeated nonces can break the authenticity of OCB
  - An attacker can combine blocks from messages authenticated with OCB to create another authenticated message
  - ...but unlike GCM, it cannot extract the underlying key!!

Efficiency

- OCB and GCM make about as many calls to the block cipher
- GCM used to be 3x slower than OCB
  - AES and GHASH competed for CPU resources
- OCB requires encryption and decryption, contrary to GCM

Synthetic IV Mode (SIV)

Strengthening AES-GCM against nonce-reuse.

Generic composition of nonce-based encryption and a PRF

- $t \leftarrow PRF(k_1, a \,||\, p \,||\, n)$
- $c \leftarrow Enc(k_2, n = t, p)$
- Output (c, t)

Synthetic IV Mode (SIV)

Strengthening AES-GCM against nonce-reuse.

Generic composition of nonce-based encryption and a PRF

- $t \leftarrow PRF(k_1, a \,||\, p \,||\, n)$
- $c \leftarrow Enc(k_2, n = t, p)$
- Output (c, t)

Q1: What happens if?

- n **repeats, but** a **or** p **changes?**

Synthetic IV Mode (SIV)

Strengthening AES-GCM against nonce-reuse.

Generic composition of nonce-based encryption and a PRF

- $t \leftarrow PRF(k_1, a \,||\, p \,||\, n)$
- $c \leftarrow Enc(k_2, n = t, p)$
- Output (c, t)

Q1: What happens if?

- n **repeats, but** a **or** p **changes?**
- **All of** a**,** n **and** p **repeat?**

Synthetic IV Mode (SIV)

Strengthening AES-GCM against nonce-reuse.

Generic composition of nonce-based encryption and a PRF

- $t \leftarrow PRF(k_1, a \,||\, p \,||\, n)$
- $c \leftarrow Enc(k_2, n = t, p)$
- Output (c, t)

Q1: What happens if?

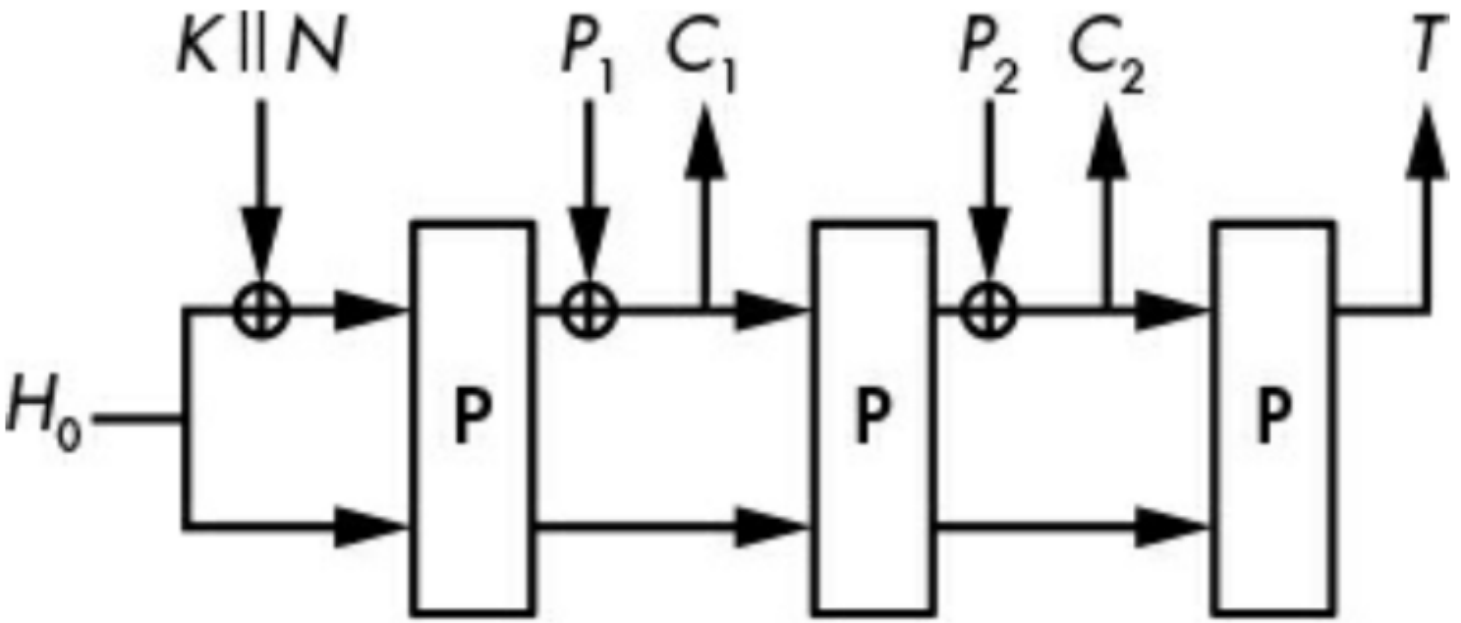
- n **repeats, but** a **or** p **changes?**
- **All of** a**,** n **and** p **repeat?**

- Tag used as encryption nonce is fresh w/ high probability
- Scheme is not streamable! **Q2: Why?**

Building AE from Permutations

Recall the sponge construction of SHA-3

Closely related construction – Duplex – gives an AEAD



- P is a fixed (unkeyed) permutation and h_0 is a public value
- Last block must be padded
- AEAD versions slightly more involved. Not covered

Permutation-based AEs

Resulting constructions are

- Fast
- Streamable

Permutation-based AEs

Resulting constructions are

- Fast
- Streamable

Interesting nonce reuse resilience:

- Unforgeability not affected
- Plaintexts compromised
  - First block
  - Subsequent block if there is a common prefix
- Plaintexts remain confidential after divergence

Key Takeaways

- Authenticated Encryption ensures confidentiality and integrity
- Encrypt-and-MAC
  - Exposed MAC. Not good!
- MAC-then-Encrypt
  - Possible decryption of malicious ciphertexts;
  - Not necessarily broken; has subtle issues
- Encrypt-then-MAC
  - The “safest” way to do it

Key Takeaways

- Authenticated Encryption ensures confidentiality and integrity
- Encrypt-and-MAC
  - Exposed MAC. Not good!
- MAC-then-Encrypt
  - Possible decryption of malicious ciphertexts;
  - Not necessarily broken; has subtle issues
- Encrypt-then-MAC
  - The "safest" way to do it
- General usage $\rightarrow$ ambitious requirements

Key Takeaways

- Authenticated Encryption ensures confidentiality and integrity
- Encrypt-and-MAC
  - Exposed MAC. Not good!
- MAC-then-Encrypt
  - Possible decryption of malicious ciphertexts;
  - Not necessarily broken; has subtle issues
- Encrypt-then-MAC
  - The "safest" way to do it
- General usage $\rightarrow$ ambitious requirements
- AES-GCM
  - AES-CTR followed by a round of Wegman-Carter
  - ChaCha20-Poly1305 similar

Key Takeaways

- Authenticated Encryption ensures confidentiality and integrity
- Encrypt-and-MAC
  - Exposed MAC. Not good!
- MAC-then-Encrypt
  - Possible decryption of malicious ciphertexts;
  - Not necessarily broken; has subtle issues
- Encrypt-then-MAC
  - The "safest" way to do it
- General usage $\rightarrow$ ambitious requirements
- AES-GCM
  - AES-CTR followed by a round of Wegman-Carter
  - ChaCha20-Poly1305 similar
- OCB
  - Offset XOR'd with AES input/output

Key Takeaways

- Authenticated Encryption ensures confidentiality and integrity
- Encrypt-and-MAC
  - Exposed MAC. Not good!
- MAC-then-Encrypt
  - Possible decryption of malicious ciphertexts;
  - Not necessarily broken; has subtle issues
- Encrypt-then-MAC
  - The "safest" way to do it
- General usage $\rightarrow$ ambitious requirements
- AES-GCM
  - AES-CTR followed by a round of Wegman-Carter
  - ChaCha20-Poly1305 similar
- OCB
  - Offset XOR'd with AES input/output
- AEAD from permutations
  - Absorb phase from Sponge; gets blocks between permutations
  - Authentication tag recovered at the end

Computer Security Foundations
Week 9: MACs and Authenticated Encryption

Bernardo Portela

L.EIC - 24