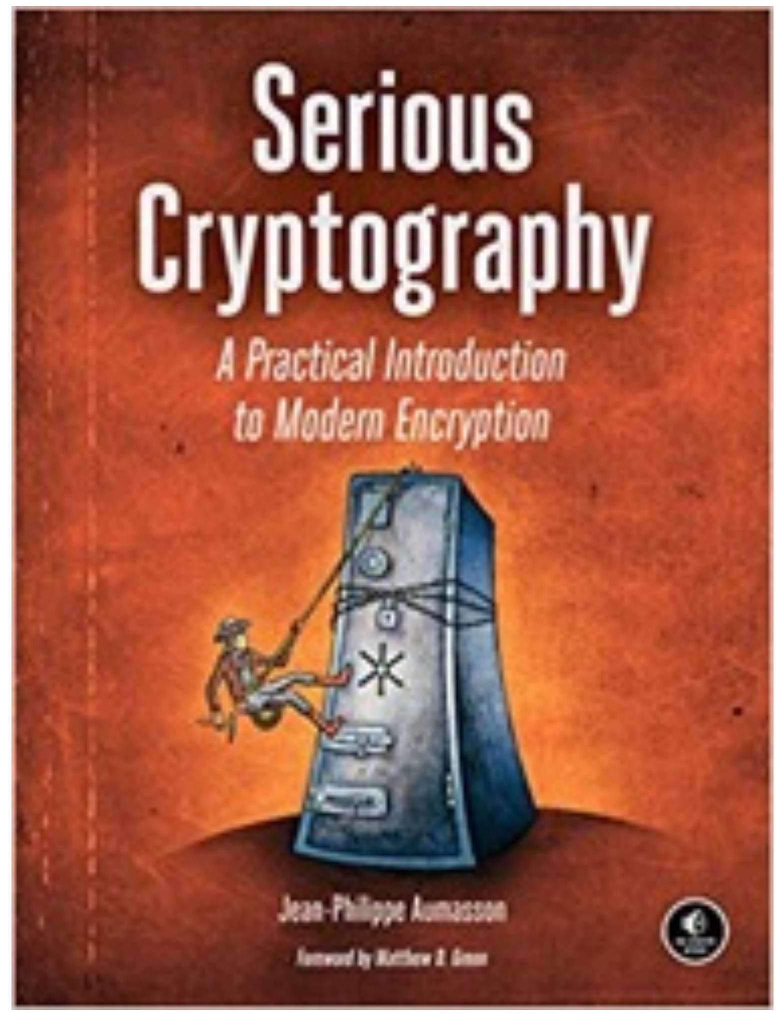
# CNIT 141

# Cryptography for Computer Networks




## 7. Keyed Hashing

**Updated 10-16-2023**

# Topics

- Message Authentication Codes (MACs)
- Pseudorandom Functions (PRFs)
- Creating Keyed Hashes from Unkeyed Hashes
- Creating Keyed Hashes from Block Ciphers: CMAC
- Dedicated MAC Designs
- How Things Can Go Wrong

# Keyed Hashing

- Anyone can calculate the SHA hash of a message
  - No secret value involved
- Keyed hashing forms the basis for two algorithms
  - **Message Authentication Code (MAC)**
  - **Pseudorandom Function (PRF)**

# Message Authentication Codes (MACs)

# MACs

- A MAC protects a message's integrity and authenticity with a tag **T**
  - **T = MAC(*K*, *M*)**
- Verifying the MAC proves both that the message wasn't altered, and that it came from the sender holding the key

# MACs in Secure Communication

- MACs are used in
  - IPSec, SSH, and TLS
- 3G & 4G telephony encrypt packets but don't use a MAC
  - An attacker can modify the packets
  - Causing static on the line

# Forgery

- Attacker shouldn't be able to create a tag without knowing the key
  - Such a M, T pair is called a ***forgery***
  - A system is ***unforgeable*** if forgeries are impossible to find

# Known-Message Attack

- An attacker passively collects messages and tags
- Tries to find the key
- This is a very weak attack

# Chosen-Message Attacks

- An attacker can choose messages that get authenticated
  - And observe the authentication tags
  - The standard model to test MAC algorithms

# Replay Attacks

- MACs are not safe from ***replay attacks***
  - To detect them, protocols include a message number in each message
  - A replayed message will have an out-of-order message number

# Pseudorandom Functions (PRFs)

# PRFs

- Use a secret key to return **PRF(*K*, *M*)**
  - Output looks random
- Key Derivation schemes use PRFs
  - To generate cryptographic keys from a master key or password
- Identification schemes use PRFs
  - To generate a response from a random challenge

# Uses of PRFs

- 4G telephony uses PRFs
  - To authenticate a SIM card
  - To generate the encryption key and MAC used during a phone call
- TLS uses a PRF
  - To generate key material from a master secret and a session-specific random value

# PRF Security

- Has no pattern, looks random
- Indistinguishable from random bits
- Fundamentally stronger than MACs
  - MACs are secure if they can't be forged
  - But may not appear random

# Creating Keyed Hashes from Unkeyed Hashes

# The Secret-Prefix Construction

- Prepend key to the message, and return
  - **Hash(*K* || *M*)**
- May be vulnerable to length-extension attacks
  - Calculating **Hash(*K* || $M_1$ || $M_2$)** from **Hash(*K* || $M_1$)**
- SHA-1 & SHA-2 are vulnerable to this, but not SHA-3

# Insecurity with Different Key Lengths

- No way to tell key from message
  - If ***K*** is **123abc** and ***M*** is **def00**
  - If ***K*** is **123a** and ***M*** is **bcdef00**
    - Result is **Hash(123abcdef00)**
- To fix this, BLAKE2 and SHA-3 include a keyed mode
  - Another fix is to include the key's length in the hash: **Hash(*L* || *K* || *M*)**

# Secret-Suffix Construction

- Tag is **Hash($M$ || $K$)**
- Prevents length-extension attack
  - If you know **Hash($M_1$ || $K$)**
  - You can calculate **Hash($M_1$ || $K$ || $M_2$)**
  - But not **Hash($M_1$ || $M_2$ || $K$)**

# Secret-Suffix Construction

- But if there's a hash collision
  - **Hash($M_1$) = Hash($M_2$)**
- The tags can collide too
  - **Hash($M_1 \| K$) = Hash($M_2 \| K$)**

# HMAC Construction

- More secure than secret prefix or secret suffix
- Used by IPSec, SSH, and TLS
  - Specifed in NIST's FIPS 198-6 standard
  - And RFC 2104

# HMAC Construction

$$\mathbf{Hash}((K \oplus opad)\ \mathbf{Hash}((K \oplus ipad)\ M))$$

- Key ***K*** is usually shorter than block size
- Uses ***opad*** (outer padding) and ***ipad*** (inner padding)
    - ***opad*** is a series of 0x5c bytes as long as the block size
    - ***ipad*** is a series of 0x36 bytes as long as the block size

# Specifying Hash Function

- Must specify, as in **HMAC-SHA256**

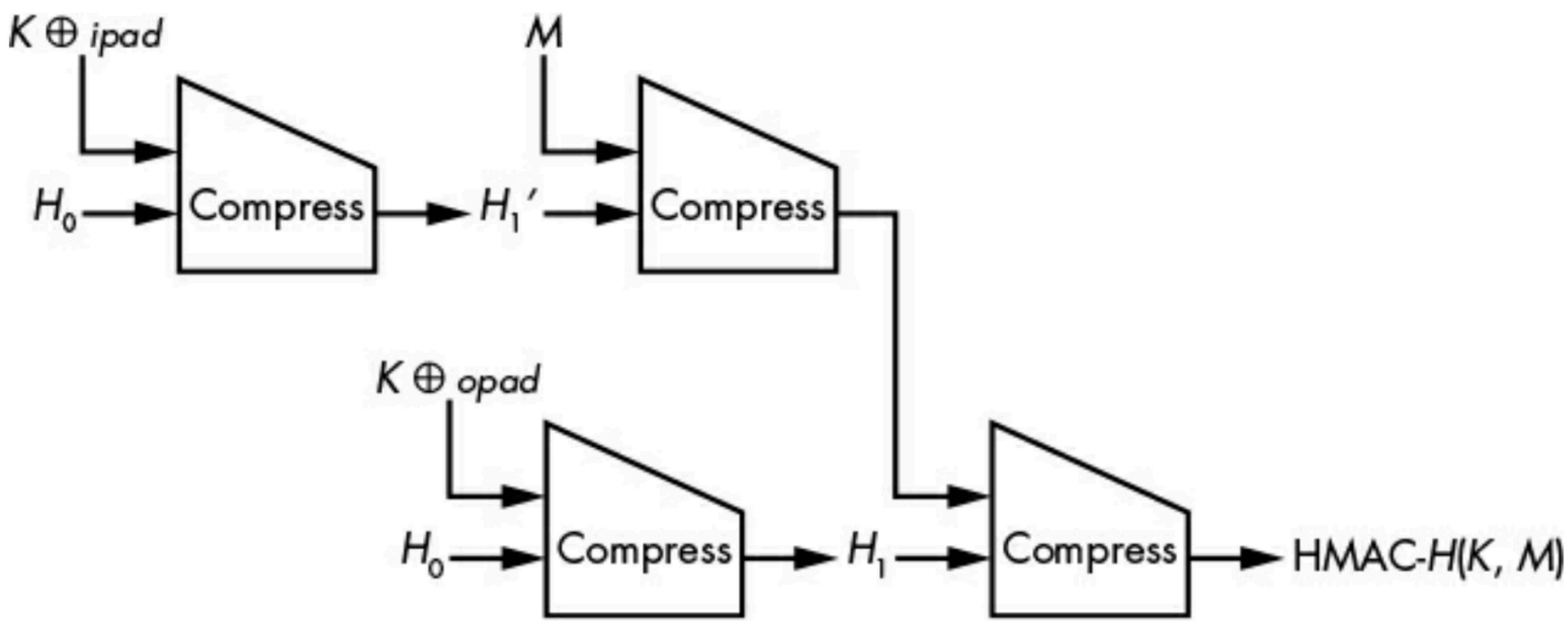



*Figure 7-1: The HMAC hash-based MAC construction*

# A Generic Attack Against Hash-Based MACs

- Can forge a HMAC tag from a hash collision
  - **Hash(K || $M_1$) = Hash(K || $M_2$)**
- Requires $\mathbf{2^{n/2}}$ calculations (digest has **n** bits)
  - **Hash(K || $M_1$ || $M_3$) = Hash(K || $M_2$ || $M_3$)**
- Works on all MACs based on an iterated hash function

# A Generic Attack Against Hash-Based MACs

- Infeasible for **n** larger than 128 bits

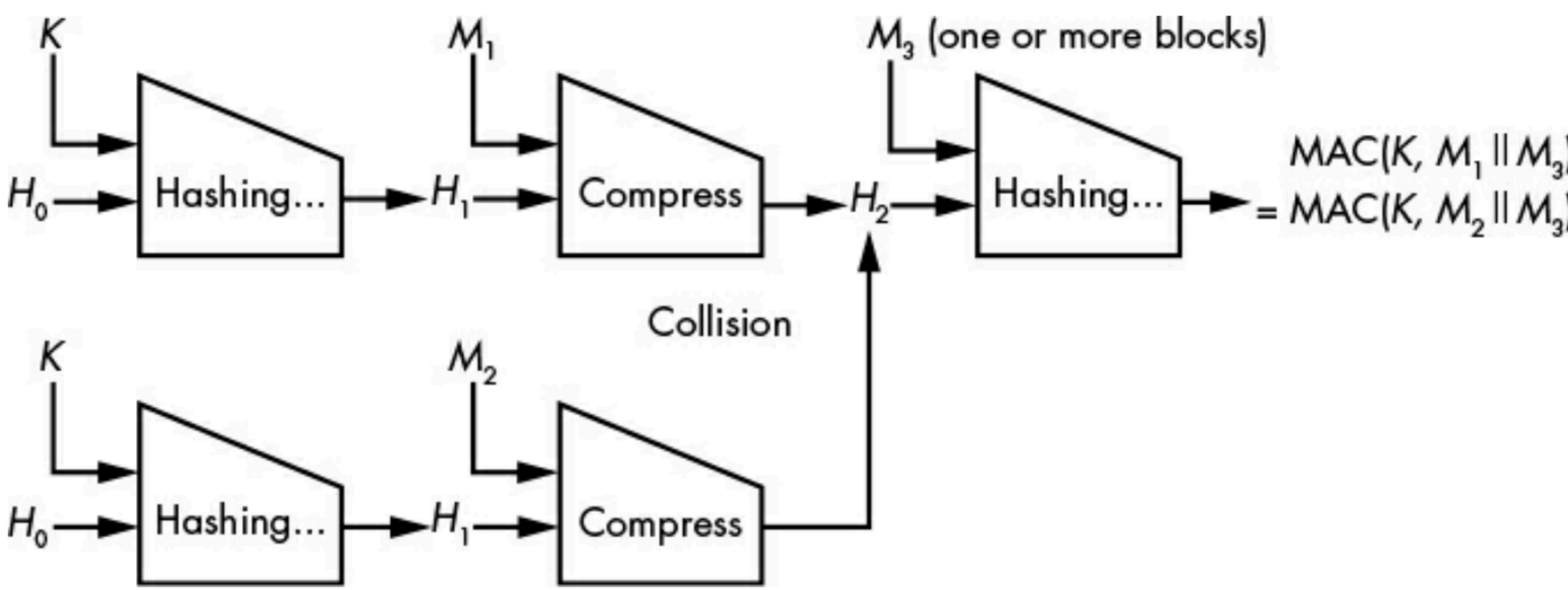



*Figure 7-2: The principle of the generic forgery attack on hash-based MACs*

Kahoot!

7a

# Creating Keyed Hashes from Block Ciphers: CMAC

# Block Ciphers

- The compression function in many hash functions is built on a block cipher
  - Ex: HMAC-SHA-256
- Next two slides from Chapter 6

# Compression-Based Hash Functions: the Merkle-Damgard Construction

- Used in MD4, MD5, SHA-1, and SHA-2
  - Also RIPEMD and Whirlpool
- $H_0$ is an initial value (IV)
- $M_1$, $M_2$, ... are blocks of message data
- The final H is the output

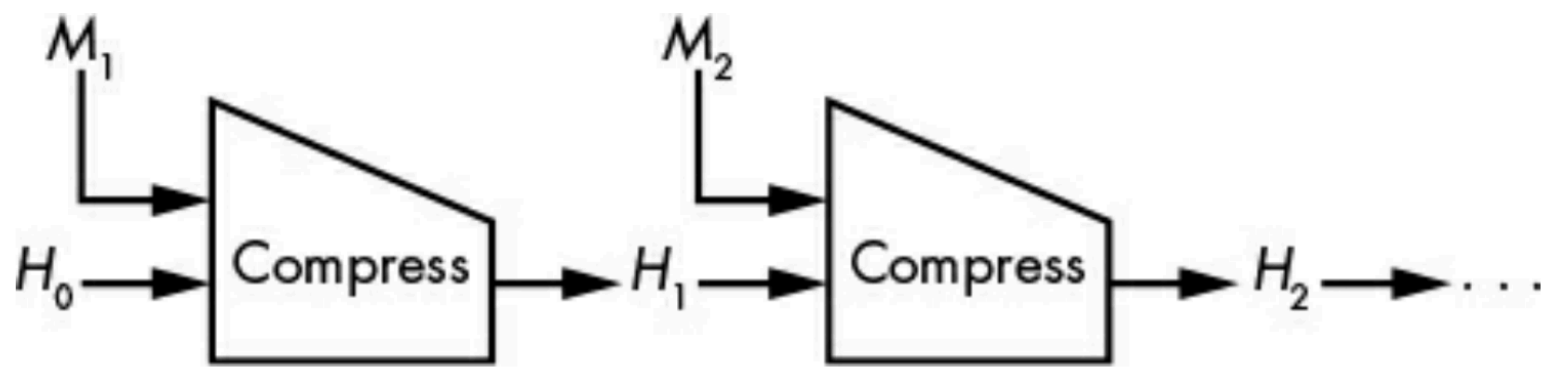



*Figure 6-4: The Merkle–Damgård construction using a compression function called Compress*

# Building Compression Functions: The Davies-Meyer Construction

- Uses a block cipher to build a compression function
- Use message blocks as keys
- The XOR feedback makes it secure against decryption

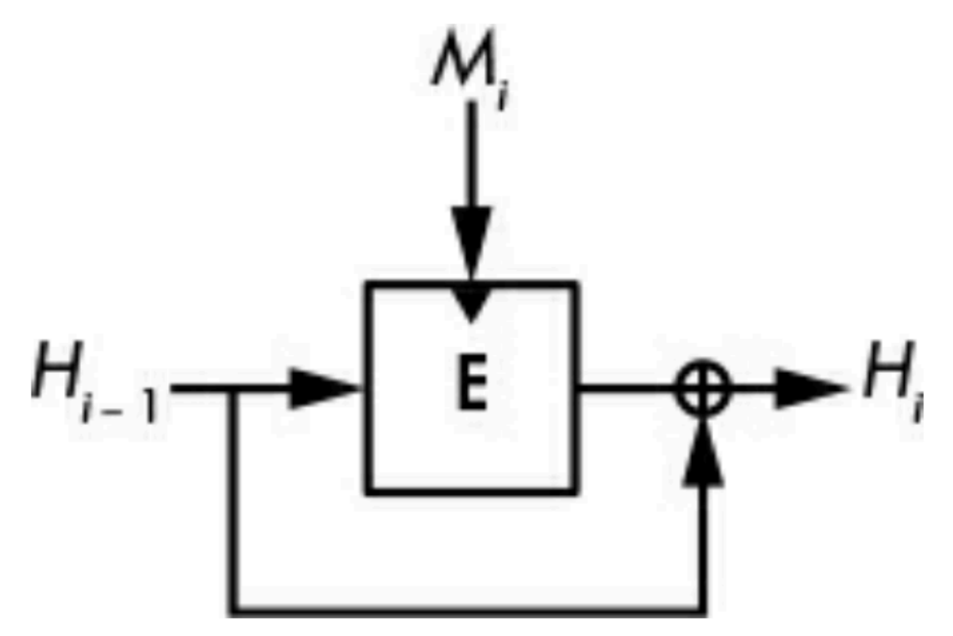


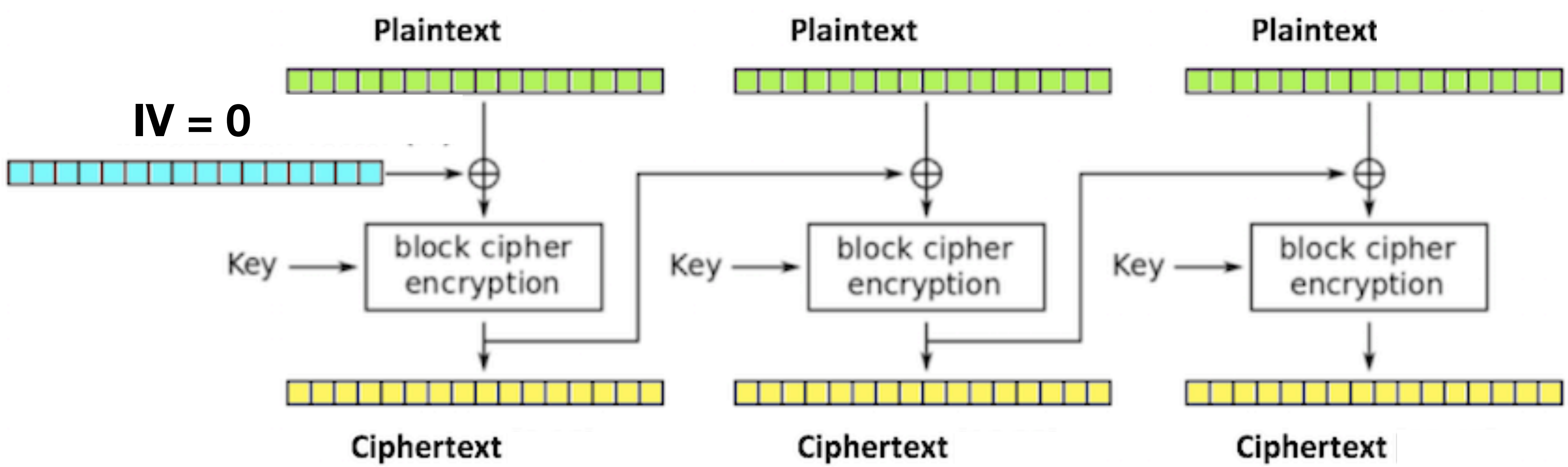

Figure 6-5: The Davies–Meyer construction. T

# CMAC and Block Ciphers

- The compression function in many hash functions is built on a block cipher
  - Ex: HMAC-SHA-256
- CMAC uses only a block cipher, such as AES
  - Less popular than HMAC
  - Used in IKE (part of IPSec)

# CBC-MAC

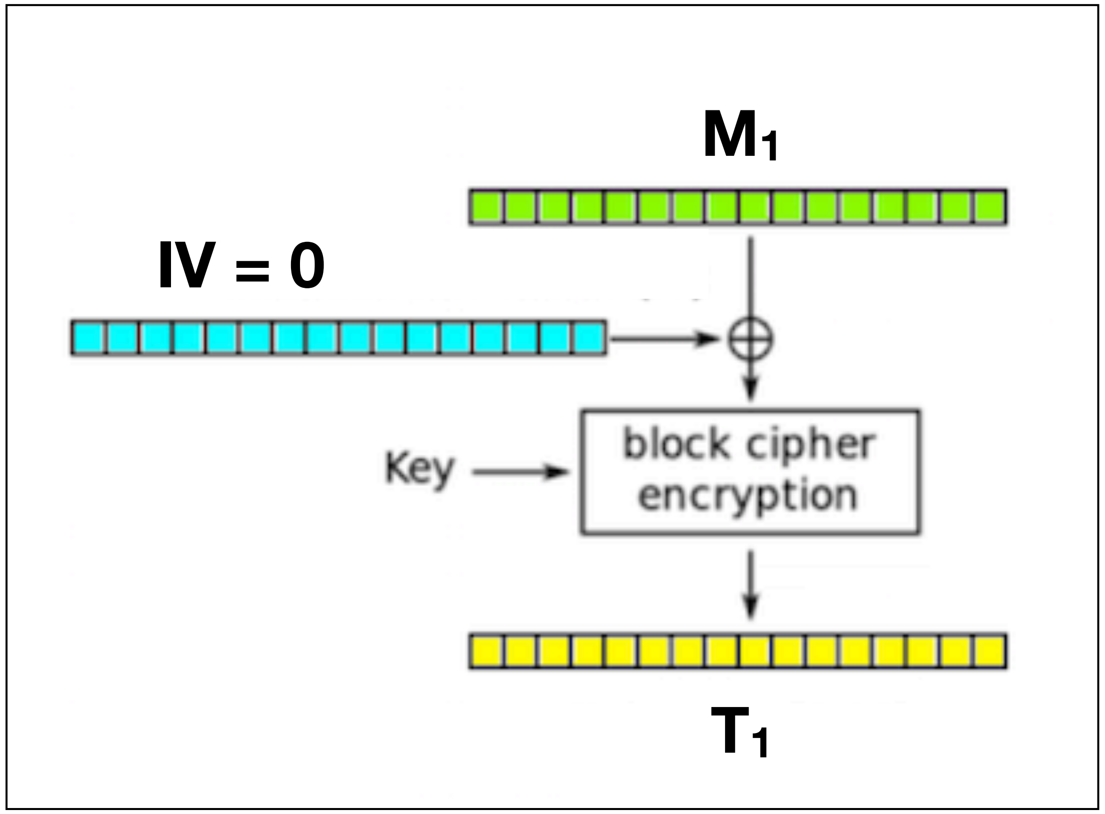
- CMAC was designed in 2005
  - As an improved version of CBC-MAC
- CBC-MAC:
  - Encrypt ***M*** with ***IV***=0
  - Discard all but the last ciphertext block

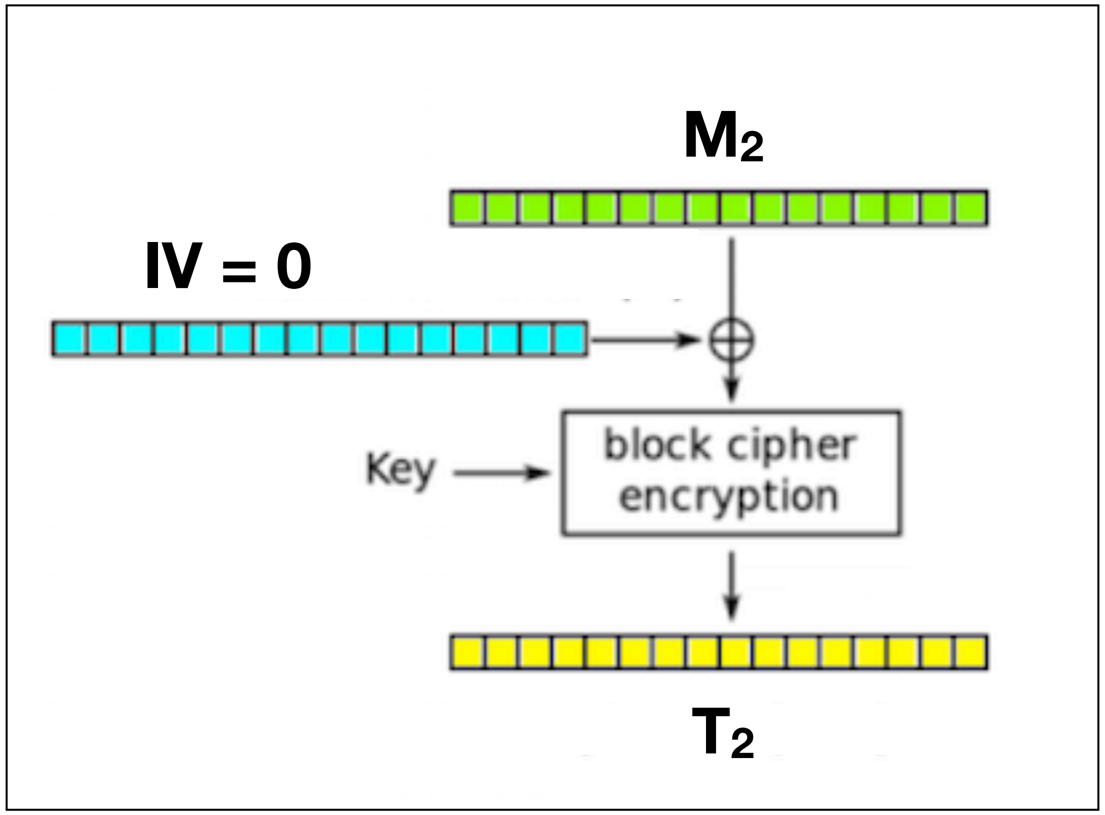



Cipher Block Chaining (CBC) mode encryption

# Breaking CBC-MAC

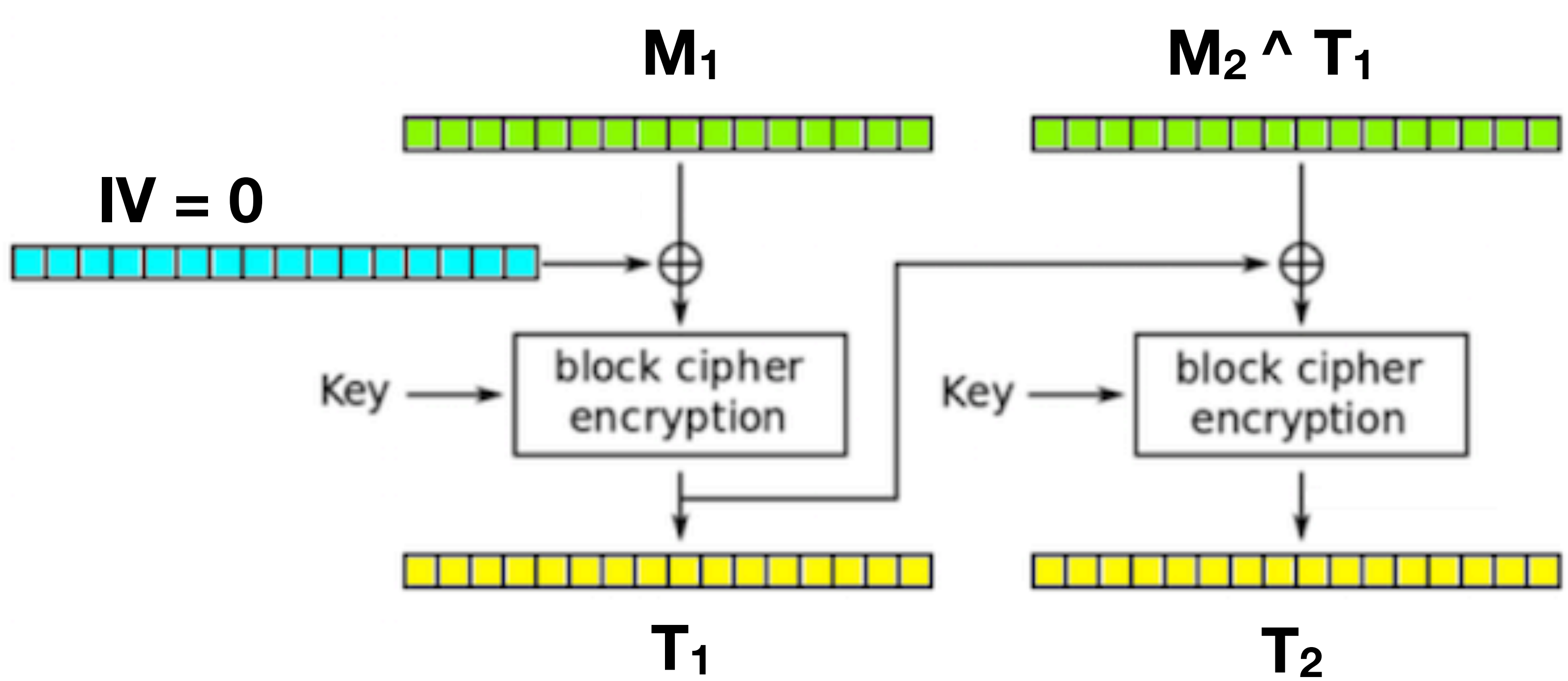
- Suppose attacker knows the tags $\mathbf{T_1}$ and $\mathbf{T_2}$
- For two single-block messages $\mathbf{M_1}$ and $\mathbf{M_2}$



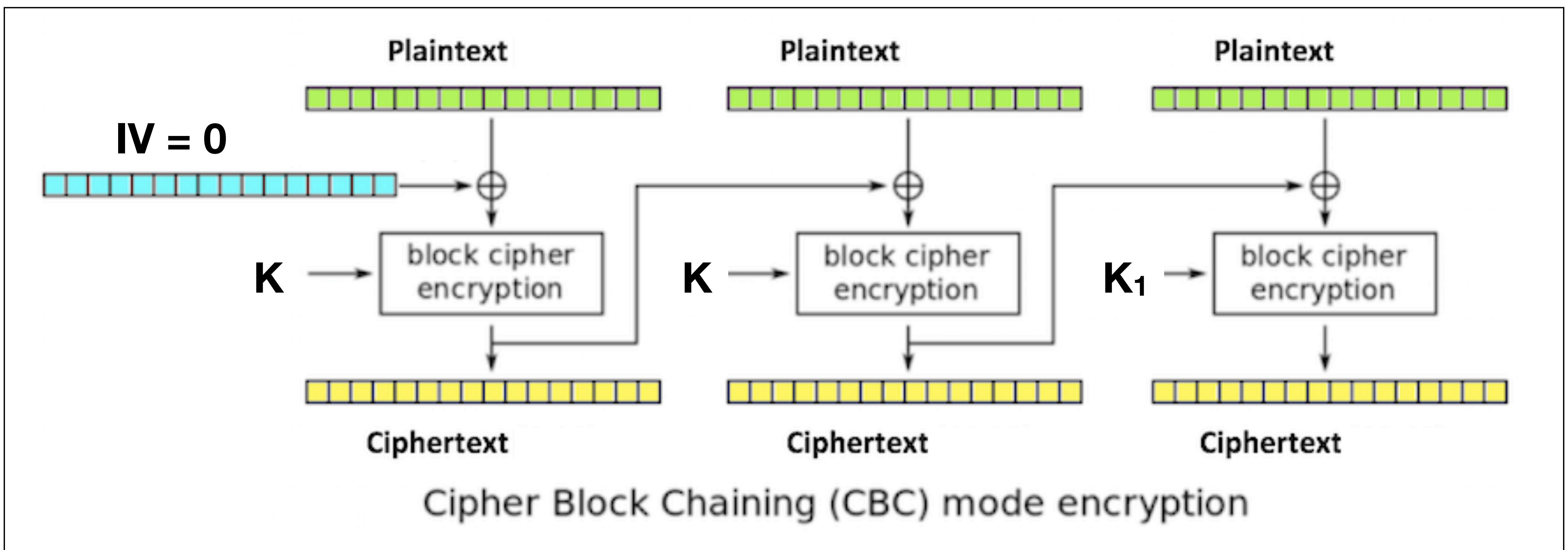
# Breaking CBC-MAC

- $\mathbf{T_2}$ is also the tag of this message:
  - $\mathbf{M_1} \,||\, (\mathbf{M_2} \wedge \mathbf{T_1})$
- For two single-block messages $\mathbf{M_1}$ and $\mathbf{M_2}$
- Attacker can forge a message and tag



# Fixing CBC-MAC

- CMAC
  - Uses key **K** to create $\mathbf{K_1}$ and $\mathbf{K_2}$
  - Encrypts last block with a different key



# CMAC

- If the message fills the last block exactly
  - Uses ***K*** and ***$K_1$***

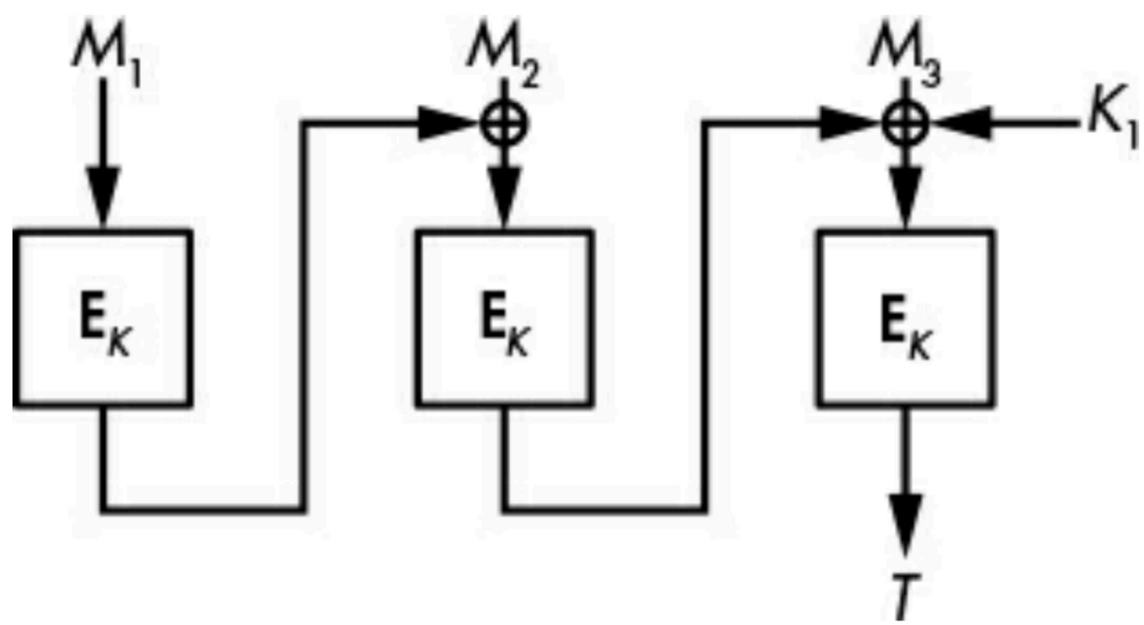



*Figure 7-3: The CMAC block cipher–based MAC construction when the message is a sequence of integral blocks*

# CMAC

- If padding is needed
  - Uses ***K*** and $\boldsymbol{K_2}$

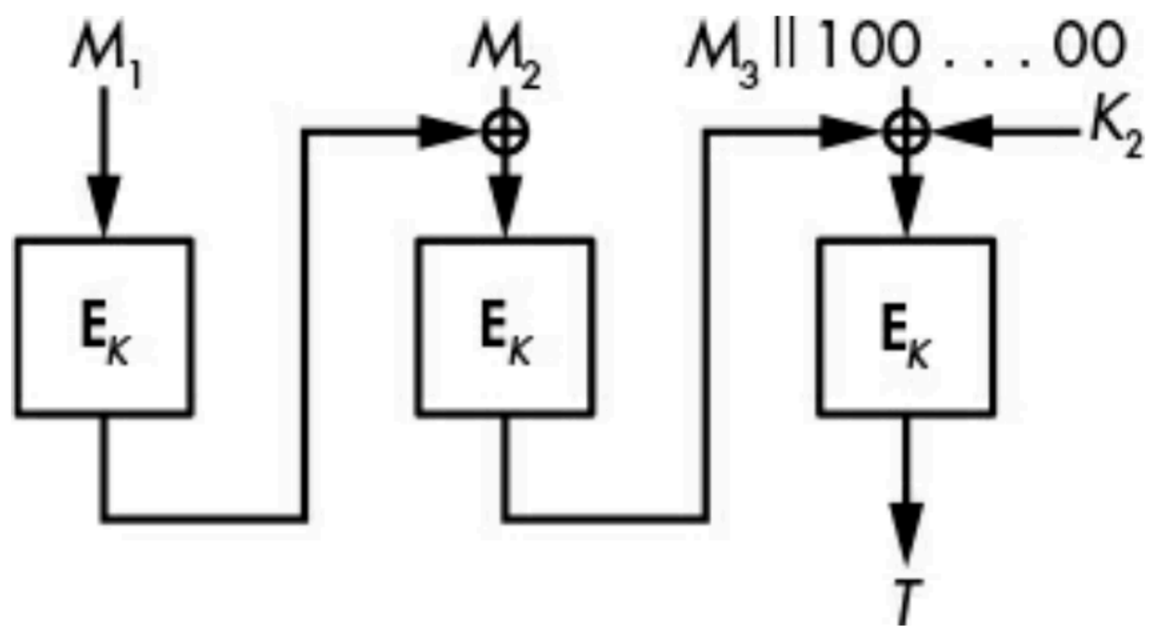



*Figure 7-4: The CMAC block cipher–based MAC construction when the last block of the*

# Dedicated MAC Designs

# Dedicated Design

- The preceding systems use hash functions and block ciphers to build PRFs
- Convenient but inefficient
- Could be made faster by designing algorithms specifically for MAC use case

# Poly1305

- Designed in 2005
- Optimized to run fast on modern CPUs
- Used by Google for HTTPS and OpenSSH

# Universal Hash Functions

- UHF is much weaker than a cryptographic hash function
  - But much faster
  - Not collision-resistant
- Uses a secret key ***K***
  - **UH(*K*, *M*)**

# Universal Hash Functions

- Only one security requirement
- For two messages $M_1$ *and* $M_2$
- Negligible probability that
  - **UH($K$, $M_1$) = UH($K$, $M_2$)**
  - For a random $K$
- Doesn't need to be pseudorandom

# Universal Hash Functions

$$\mathbf{UH}(R, K, M) = R + M_1K + M_3K^2 + M_3K^3 + \dots + M_nK^n \ \mathbf{mod}\ p$$

- Weakness:
- ***K*** can only be used once
- Otherwise an attacker can solve two equations like this and gain information about the key

# Wegman-Carter MACs

$$\mathbf{MAC}(K_1, K_2, N, M) = \mathbf{UH}(K_1, M) + \mathbf{PRF}(K_2, N)$$

- Builds a MAC from a universal hash function and a PRF
  - Using two keys $K_1$ and $K_2$
  - And a nonce $N$ that is unique for each key, $K_2$

# Wegman-Carter MACs

- Secure if
  - **UH is a secure universal hash.**
  - **PRF is a secure PRF.**
  - **Each nonce $N$ is used only once for each key $K_2$.**
  - **The output values of UH and PRF are long enough to ensure high enough security.**

# Poly1305-AES

$$\textbf{Poly 1305}(K_1, M) + \textbf{AES}(K_2, N) \textbf{ mod } 2^{128}$$

- Much faster than HMAC-based MACSs or even CMACs
  - Only computes one block of AES
  - Poly1305 is a universal hash
  - Remaining processing runs in parallel with simple arithmetic operations
- Secure as long as AES is

# SipHash

- Poly1305 is optimized for long messages
  - Requires nonce, which must not be repeated
  - For small messages, Poly1305 is overkill
- SipHash is best for short messages
  - Less than 128 bytes

# SipHash

- Designed to resist DoS attacks on hash tables
- Uses XORs, additions, and word rotations

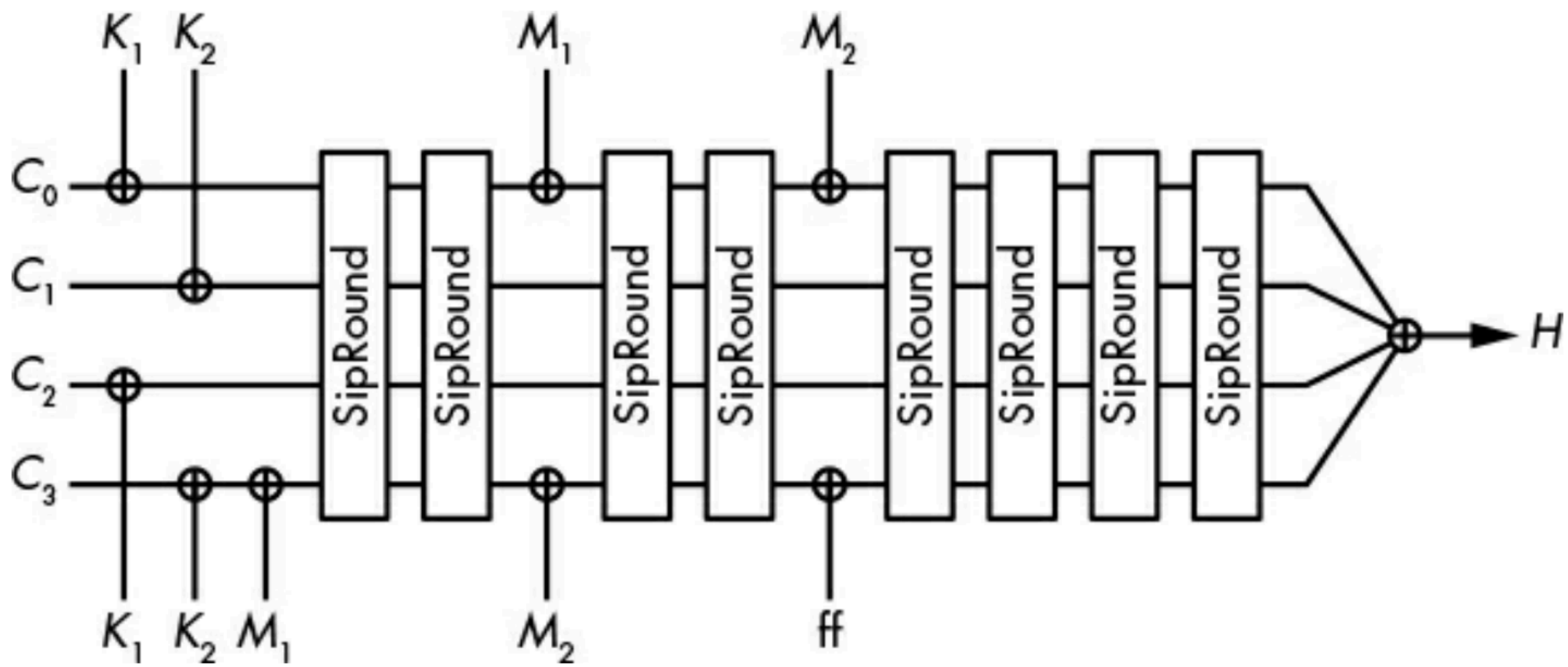



*Figure 7-5: SipHash-2-4 processing a 15-byte message (a block,* $M_1$, of 8 bytes and a block, $M_2$, of 7 bytes, plus 1 byte of padding)

# How Things Can Go Wrong

# Timing Attacks on MAC Verficiation

- ***Side-channel attacks***
  - Target the implementation
  - Not the algorithm
- This code will return faster if the first byte is incorrect
- Solution: write ***constant-time*** code

```
def compare_mac(x, y, n):
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True
```

# When Sponges Leak

- If attacker gets the internal state
  - Through a side-channel attack
- Permutation-based algorithms fail
  - Allowing forgery
- Applies to SHA-3 and SipHash
- But not compression-function-based MACs
  - Like HMAC-SHA-256 and BLAKE2

Kahoot!

**7b**