ECE 382N-Sec (FA25):

L7: Memory Encryption and Integrity Protection

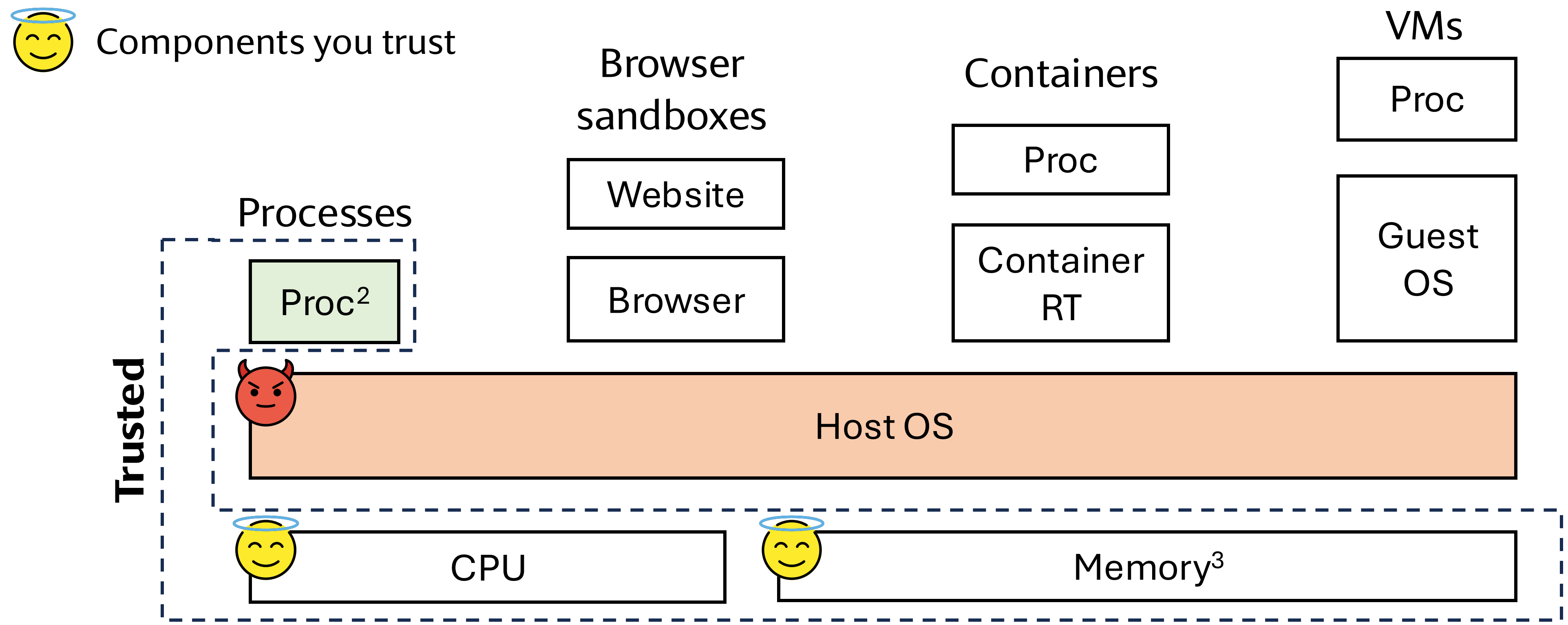
Neil Zhao

neil.zhao@utexas.edu

Before We Start

- Building Trusted-Execution Environments often involves various crypto tools
- This course focuses on general crypto primitives instead of specific algorithms and their implementations
  - These primitives are nice “hammers” to system builders
  - How these hammers are built is fascinating, but it’s out-of-scope for this course
- Our discussion simplifies certain aspects of these crypto primitives. It is good for building an intuitive understanding, but please do consult and follow various crypto standards for anything serious. Don’t re-invent the hammer!
- **A good reference:** “Serious Cryptography: A Practical Introduction to Modern Encryption” by Jean-Philippe Aumasson

Trusted-Execution Environments (TEE)[1]



[1]TEE is a somewhat overloaded term. We focus on hardware-based TEEs

[2]The process may be divided into trusted and untrusted parts

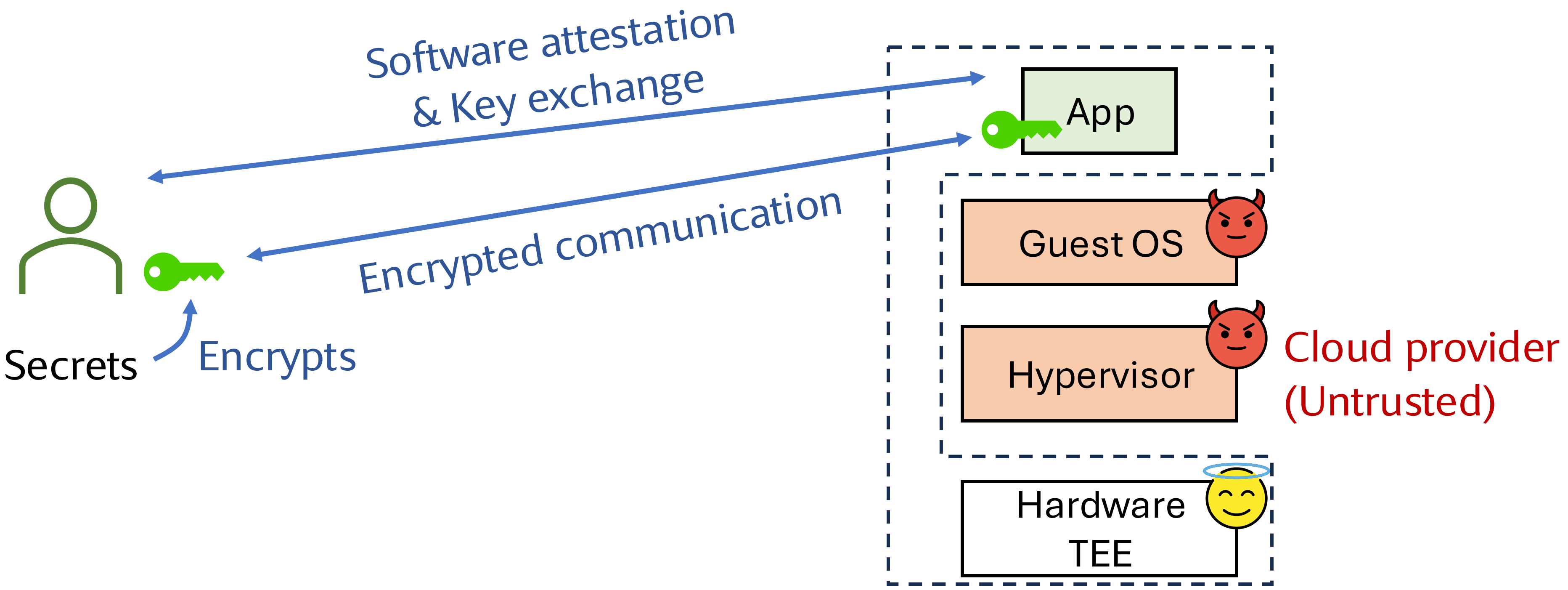
[3]Depending on the memory type and threat model, it may or may not be trusted

(Common*) Security Goals of TEEs

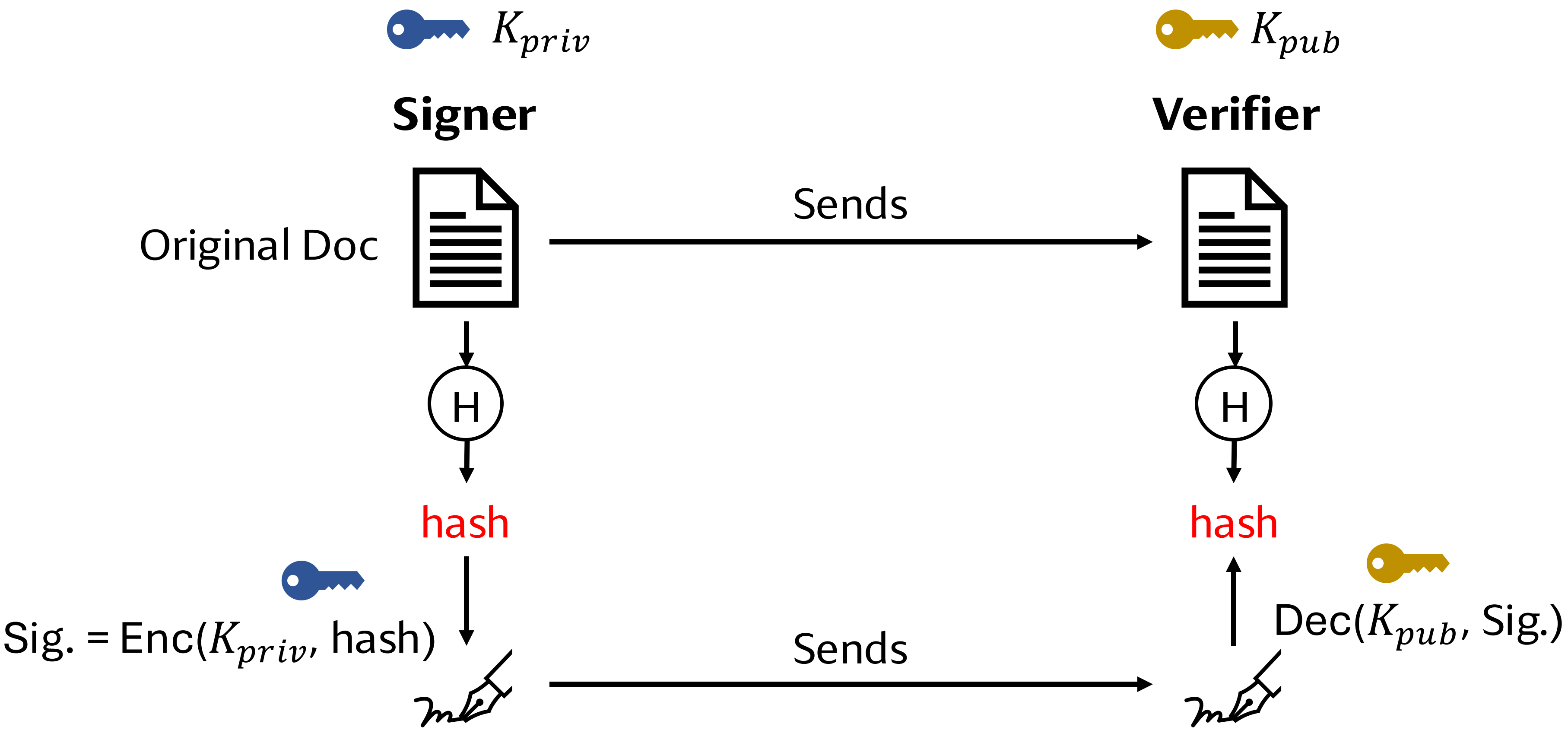
			Example Attacks	
			Software Attack	**Physical Attack**
✓	**Confidentiality**	Attacker cannot directly access my private program states (Side channel? Spectre?)	OS reads my pages	Bus snooping
✓	**Integrity**	Attacker cannot tamper with my program states (**Freshness:** Program state is up-to-date)	OS writes my pages	? Bus spoofing
✗	**Availability**	Attacker refuses to execute or give enough resources to my program	OS allocates no CPU time	Pull the plug

*Many variants exist

Before We Send Our Secrets



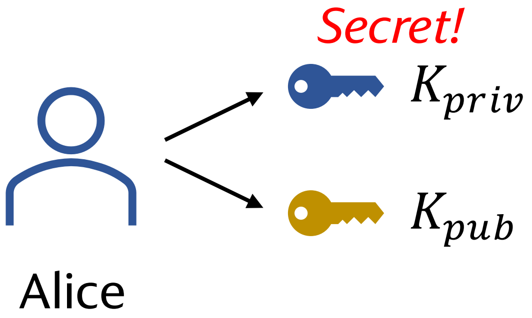
Digital Signature



Actual schemes are more complex than this

A Certificate is Like an ID Card

It binds the subject’s identity to their public key (or appearance)



Certificate

Subject Identity = Alice
Subject’s Public Key = K_{pub}
Valid From/Until = ...
Certificate Usage = ...
...
Issuer’s Public Key = $K_{pub}{}'$
Certificate Signature

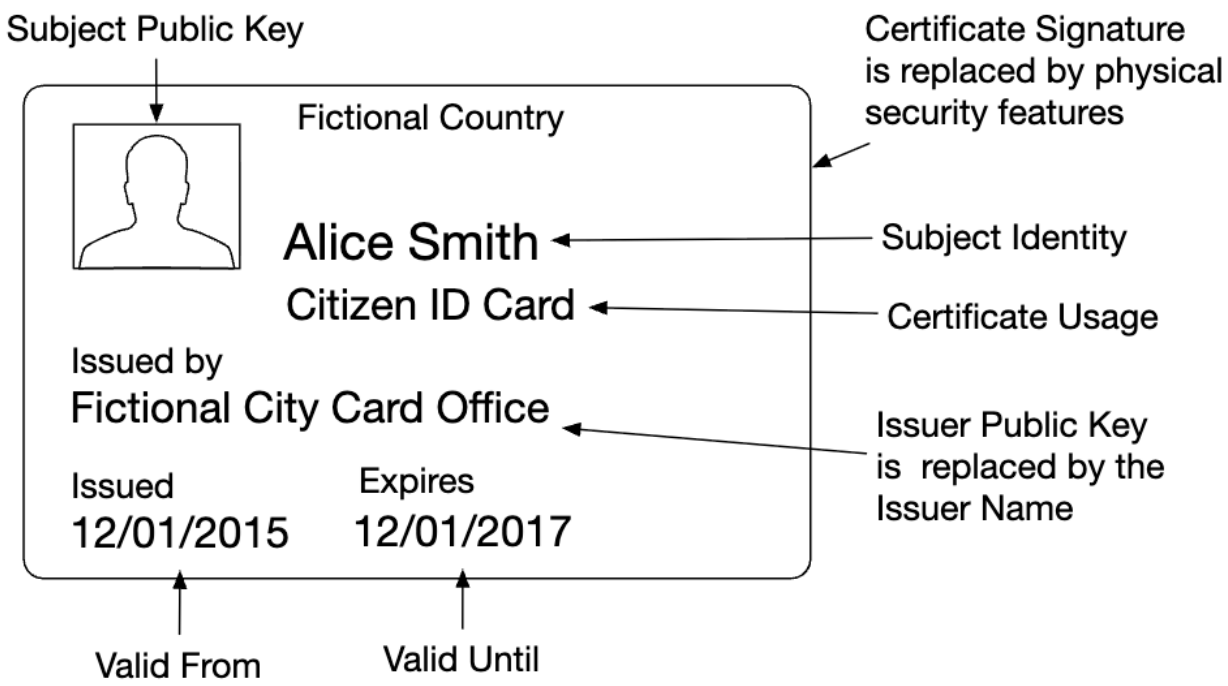
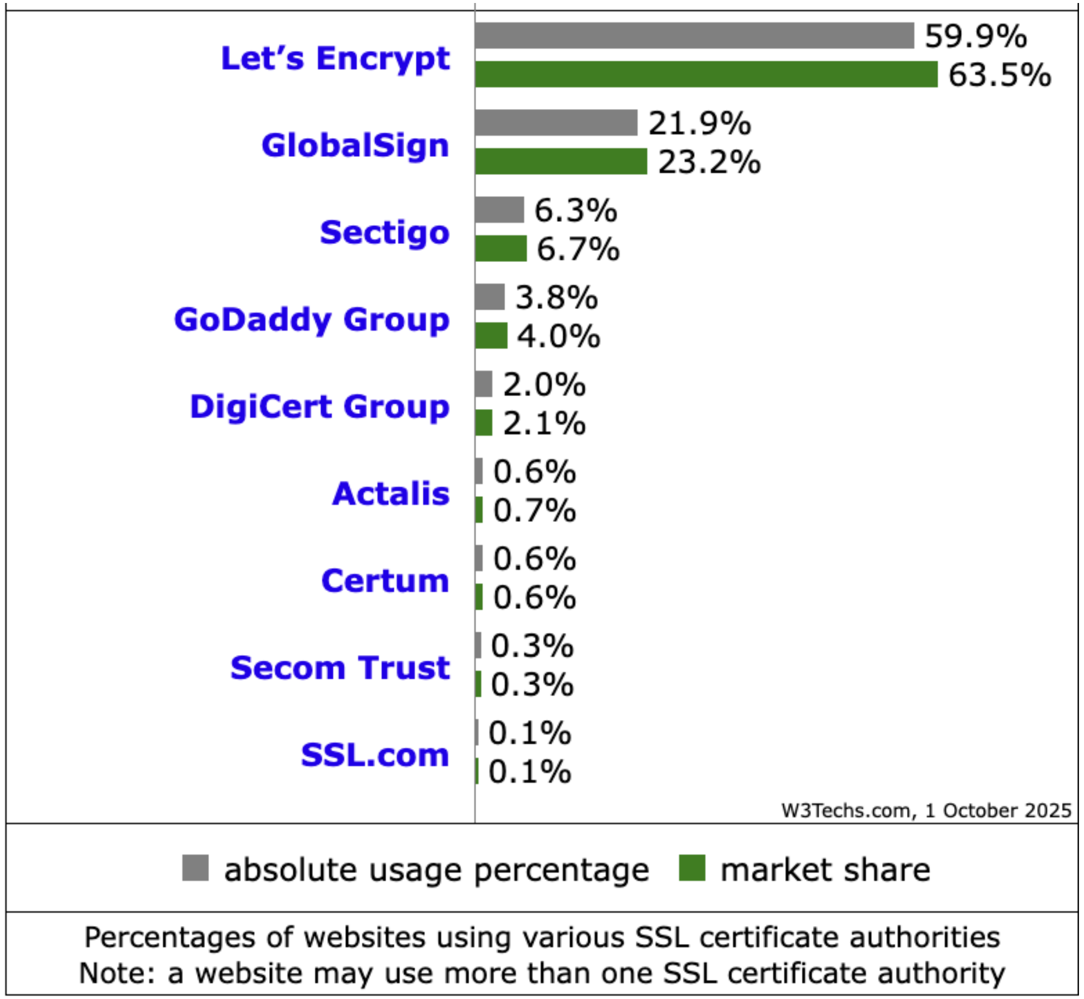


Illustration from “Intel SGX Explained” by Costan et al.

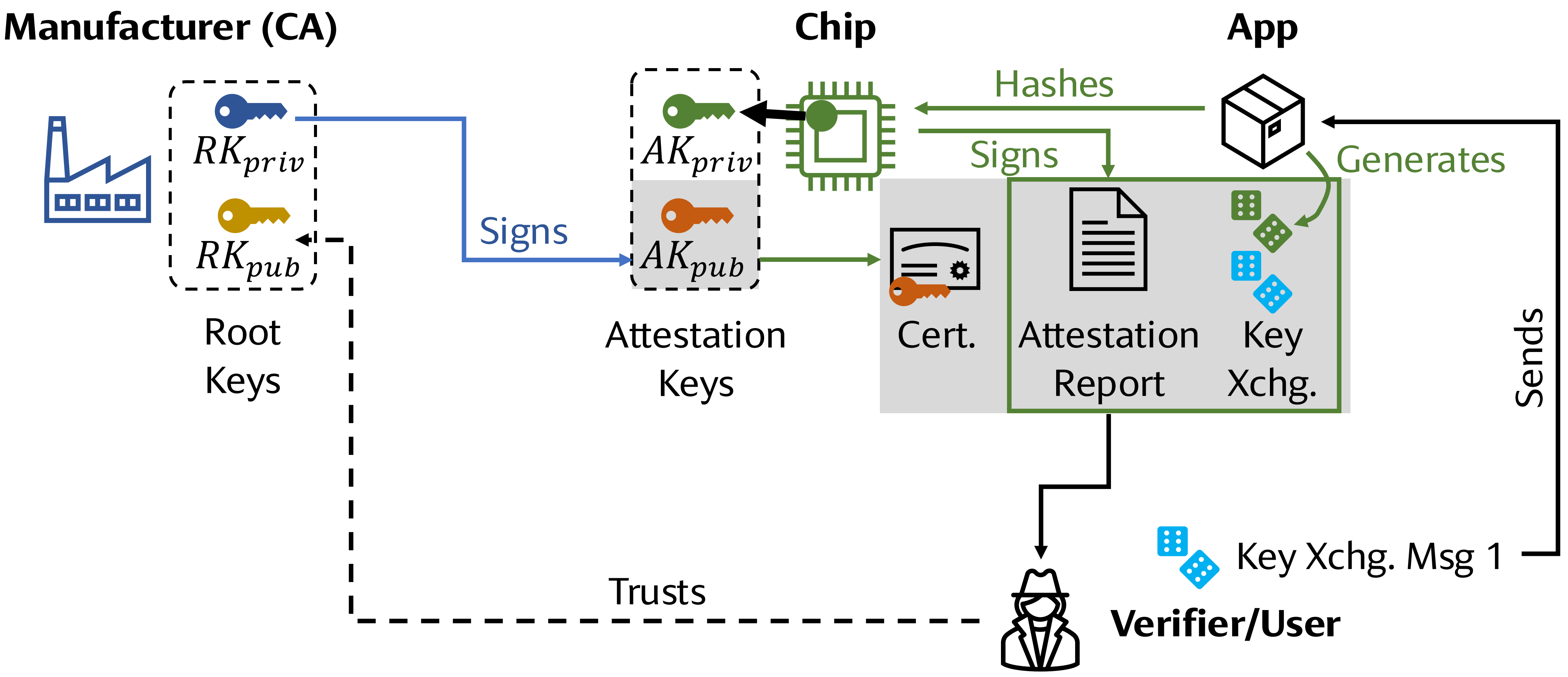
It’s a proof of identity-pubkey binding, not proof of identity

Popular CAs

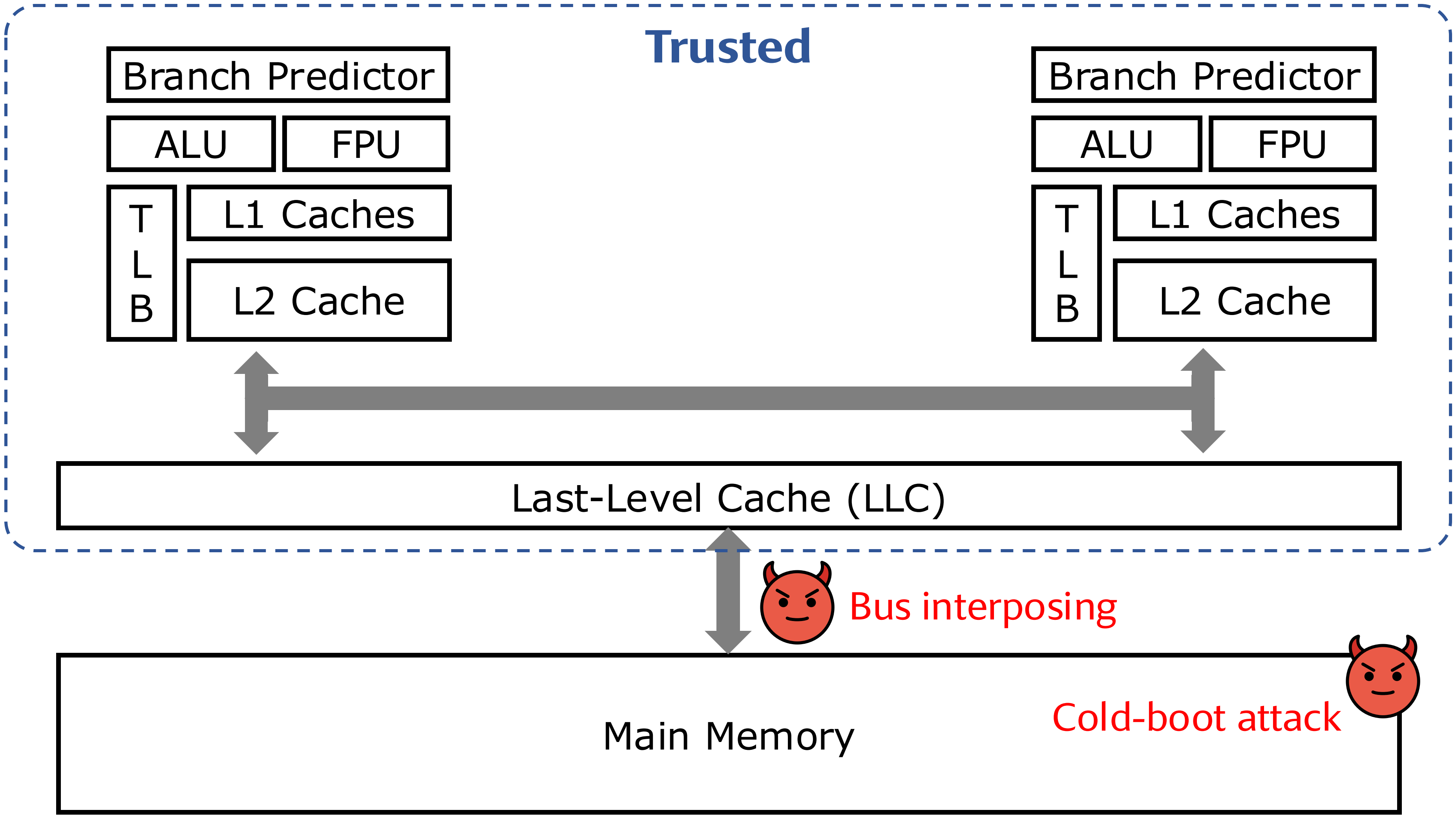


Source: https://w3techs.com/technologies/overview/ssl_certificate

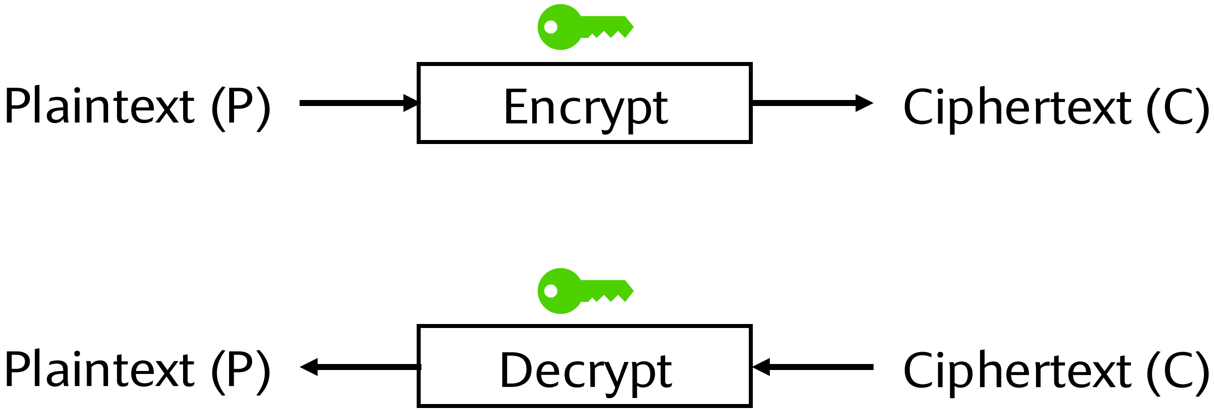
Software Attestation



The Need for Memory Encryption and Integrity Protection



Hammer 4: Symmetric Cipher



The permutation is determined by the key and informally,
the permutation should look random

Hammer 4: Symmetric Cipher

In general, we want different ciphertexts if we encrypt the same plaintext twice

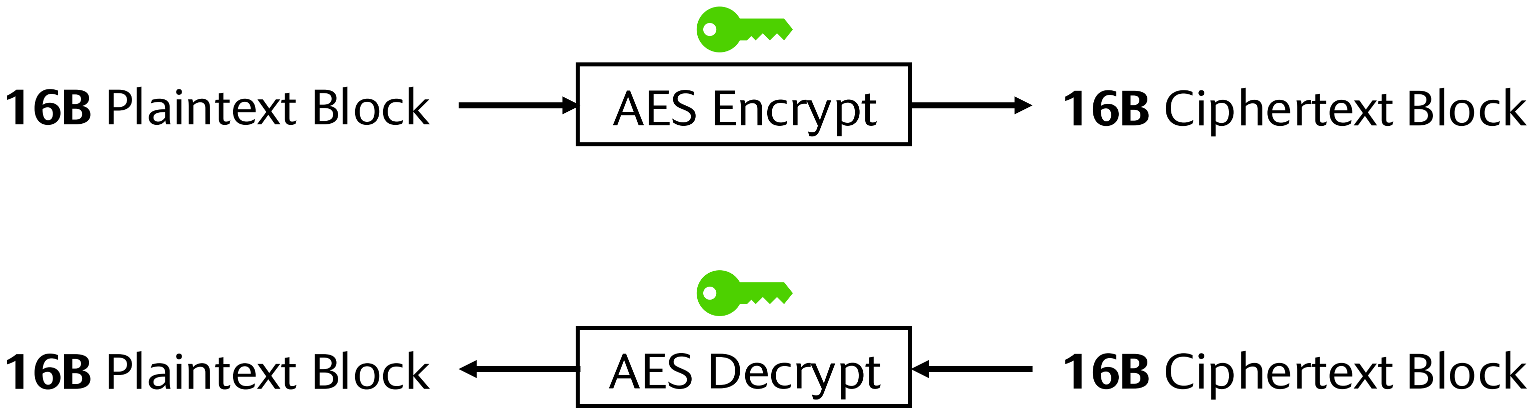
Disease		**Disease (Encrypted)**
Flu		`C#@husd`
Flu	→	`C#@husd`
Diabetes	Pad to the same length and encrypt using the same key	`yv07*we`
Covid		`fgh8973`
Flu		`C#@husd`
Covid		`fgh8973`

One-Time Pad

Plaintext (N bits)	0	1	0	0	0	0	0	1	1 ...
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Key (N random bits)	0	0	1	0	1	1	0	0	1 ...
	↓	↓	↓	↓	↓	↓	↓	↓	↓
Ciphertext (N bits)	0	1	1	0	1	1	0	1	0 ...
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Same Key	0	0	1	0	1	1	0	0	1 ...
	↓	↓	↓	↓	↓	↓	↓	↓	↓
Plaintext (N bits)	0	1	0	0	0	0	0	1	1 ...

Advanced Encryption Standard (AES)

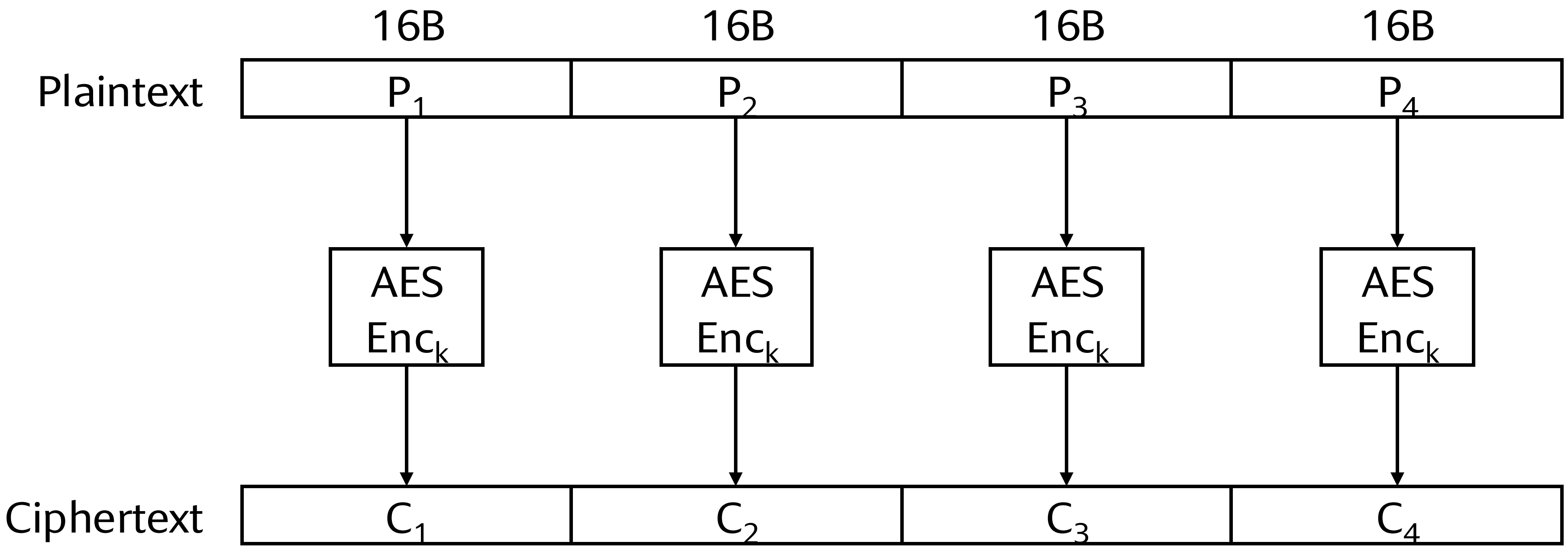
AES is a popular block cipher



Many high-end processors have hardware-accelerated AES instructions

How to use AES to encrypt a message of any length?

Electronic Codebook (ECB) Mode



Same plaintext blocks are encrypted into the same ciphertext block!

Electronic Codebook (ECB) Mode

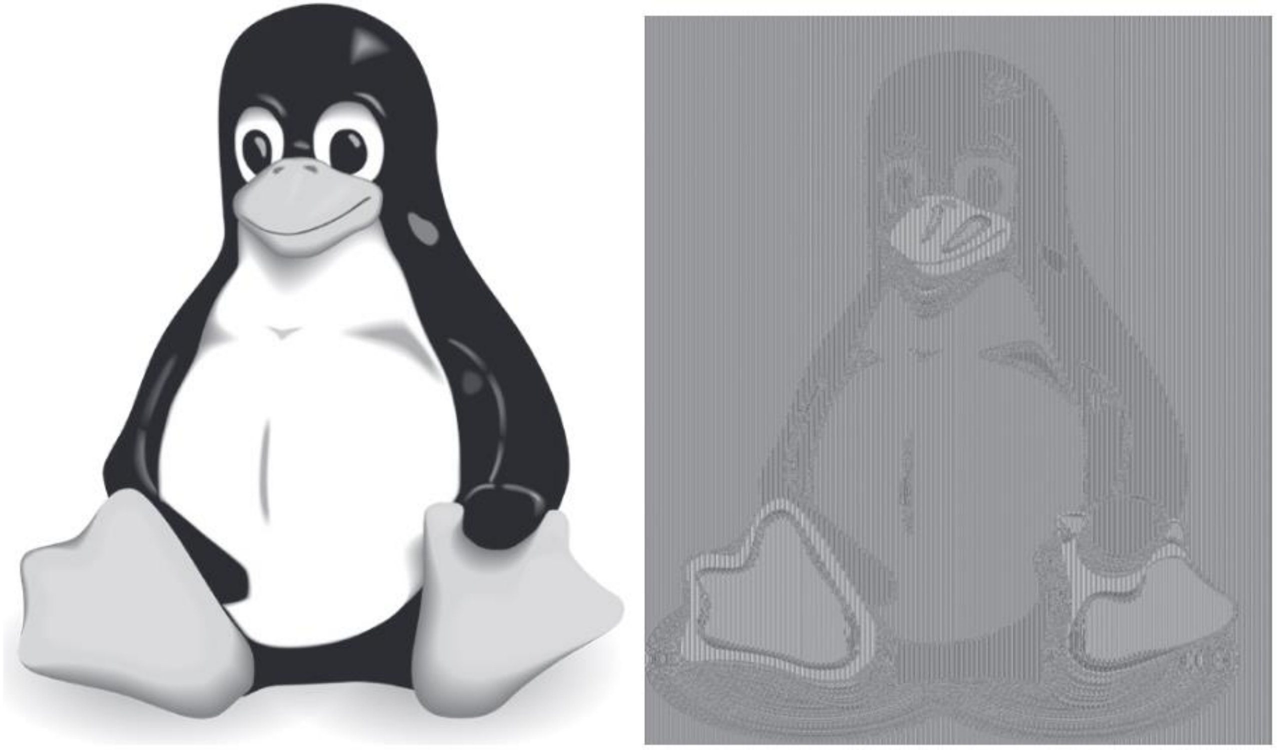
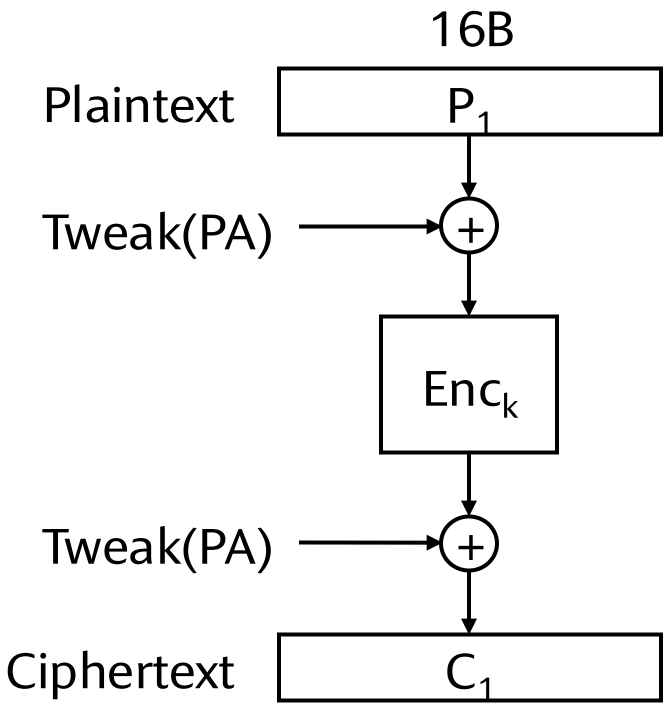


Figure 4-7: The original image (left) and the ECB-encrypted image (right)

Source: "Serious Cryptography: A Practical Introduction to Modern Encryption" by Jean-Philippe Aumasson

XOR-Encrypt-XOR (XEX) Mode

The encryption depends on the physical address (PA) of the data block



$$C = \mathrm{Enc}_k(P \oplus \mathrm{Tweak}(PA)) \oplus \mathrm{Tweak}(PA)$$

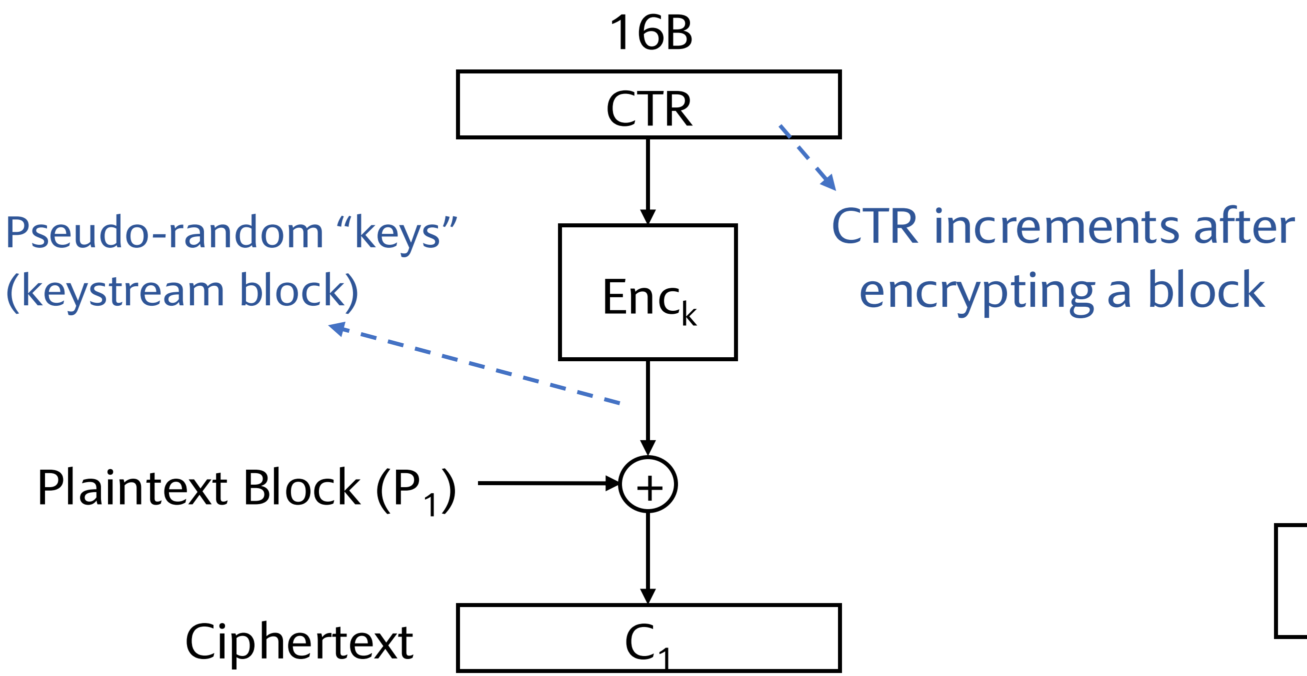
Achieve spatial uniqueness---i.e., the same plaintext block at different PAs are encrypted to different ciphertext blocks

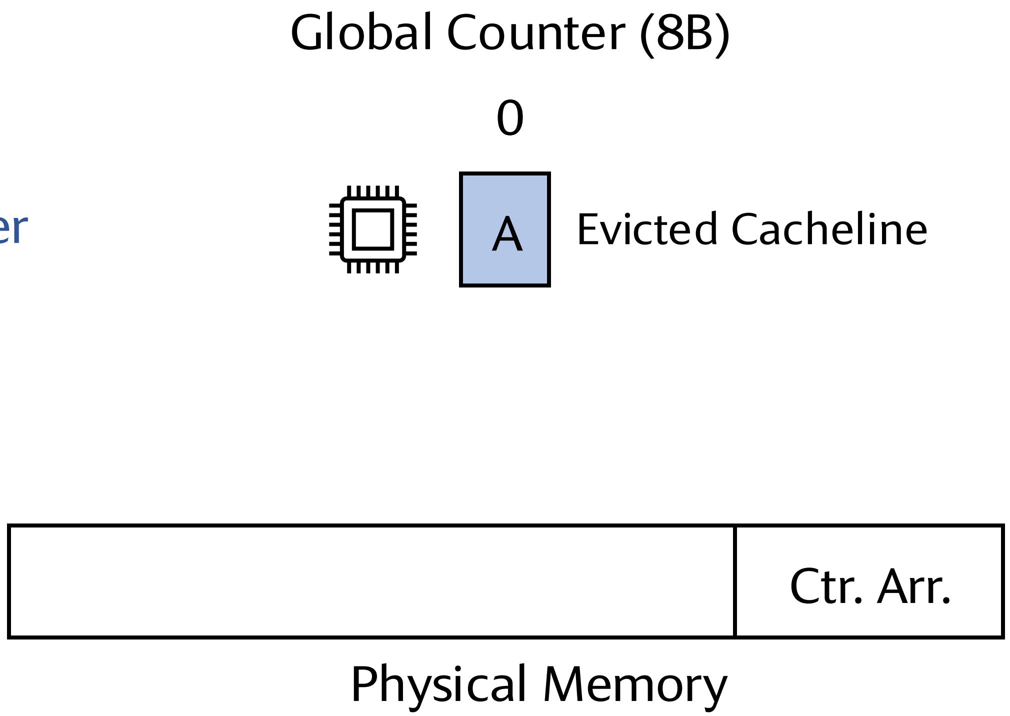
Deterministic encryption at a given location

Attacks on AMD SEV(-SNP):

- Li et al, CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel, USENIX'21
- Li et al, A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP, S&P'22

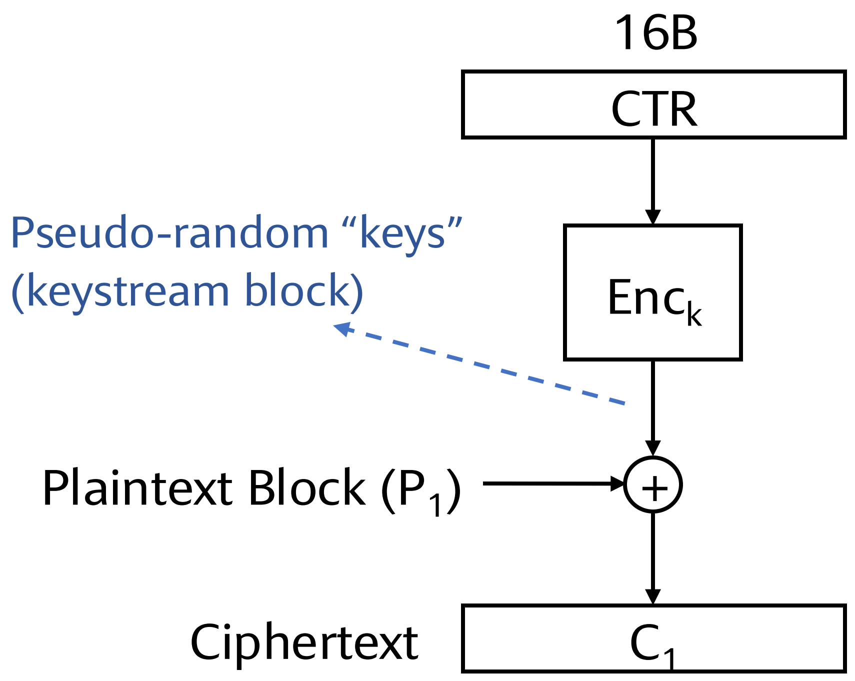
Counter (CTR) Mode



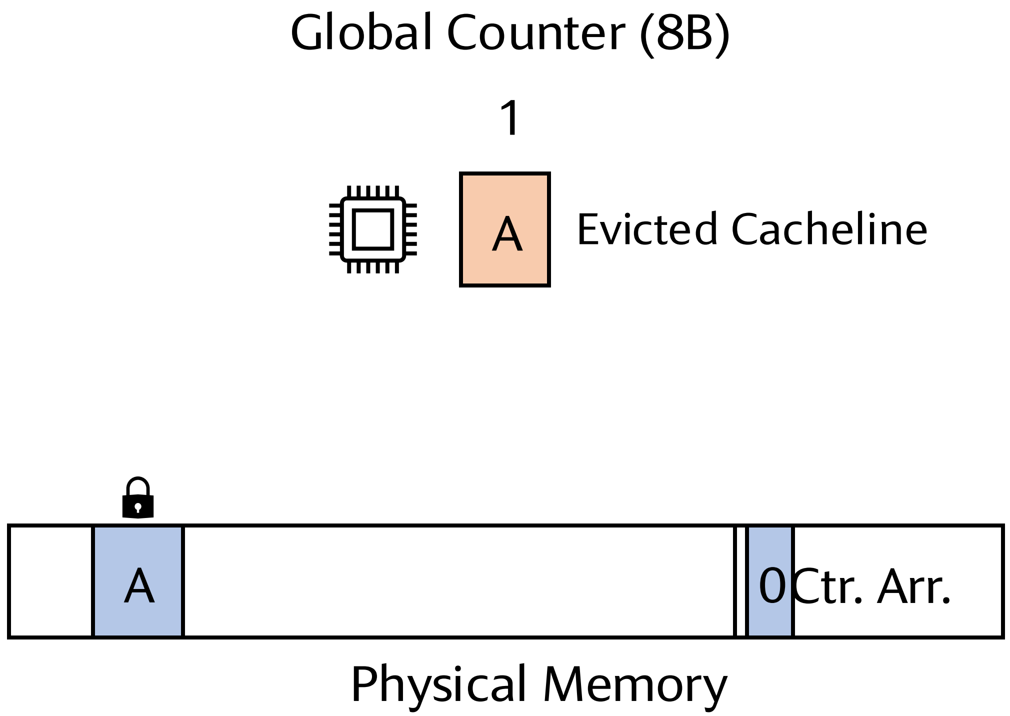


$C = \mathrm{Enc_k}(CTR) \oplus \mathrm{P}$ $\quad$ $P = \mathrm{Enc_k}(CTR) \oplus \mathrm{C}$

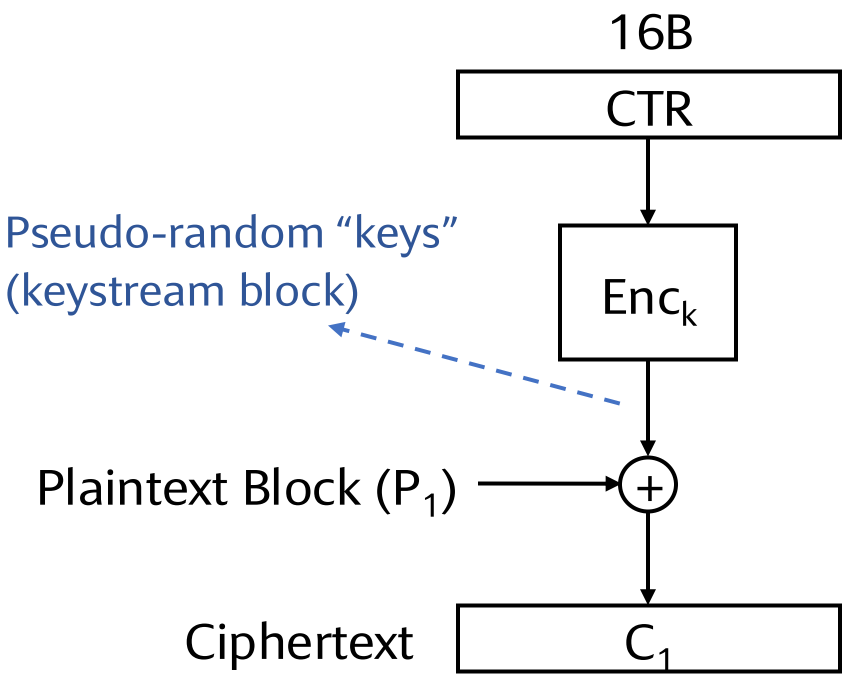
Counter (CTR) Mode



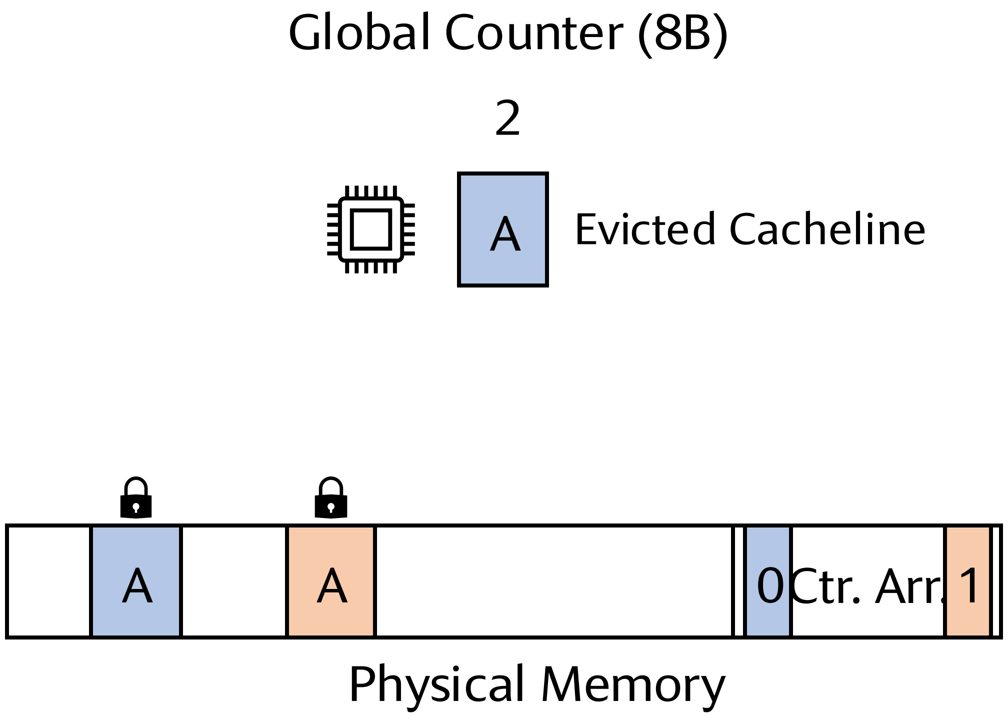
$$C = \mathrm{Enc}_k(CTR) \oplus P \qquad P = \mathrm{Enc}_k(CTR) \oplus C$$



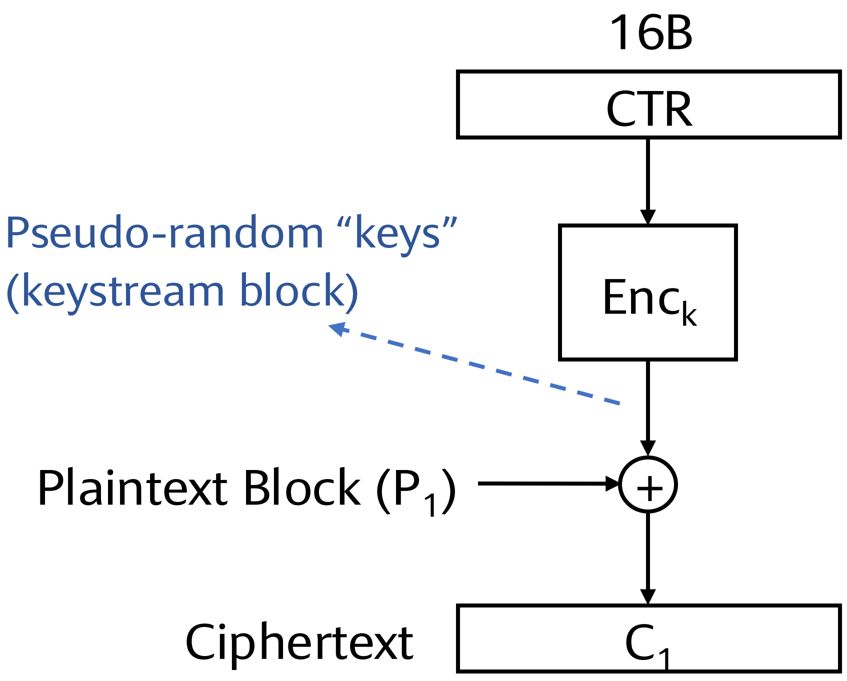
Counter (CTR) Mode



$$C = \text{Enc}_k(CTR) \oplus P \qquad P = \text{Enc}_k(CTR) \oplus C$$

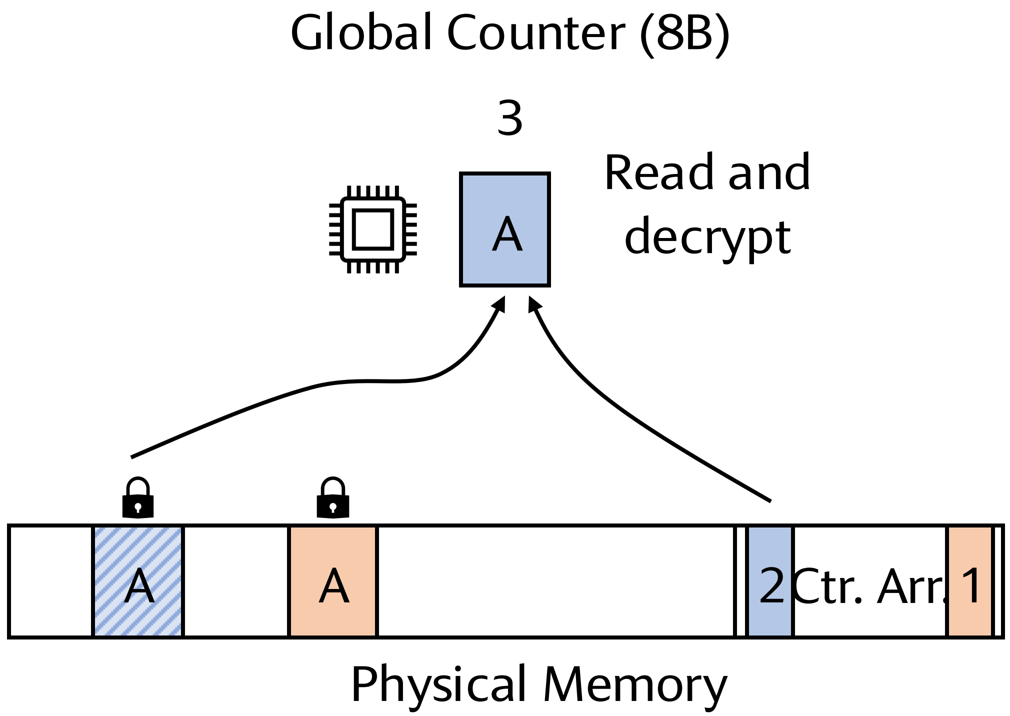


Counter (CTR) Mode



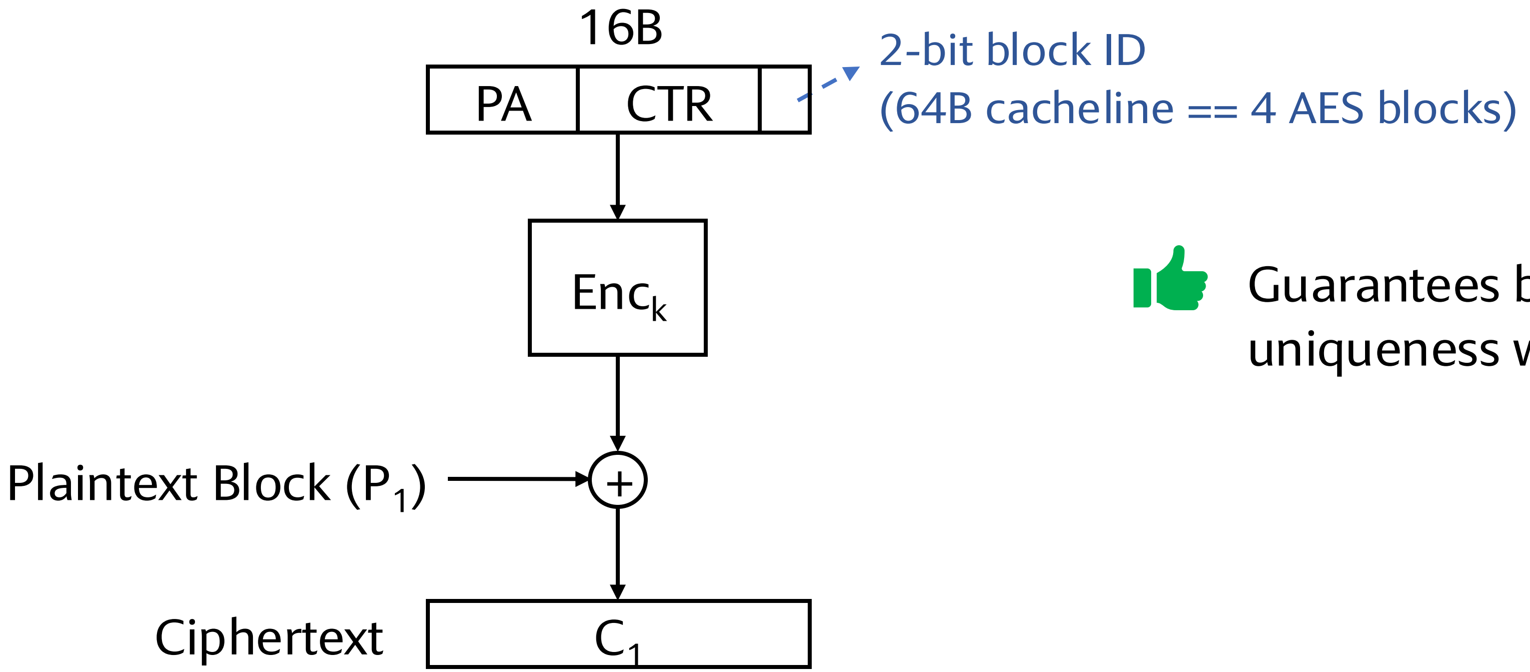
$C = \mathrm{Enc}_k(CTR) \oplus P \qquad P = \mathrm{Enc}_k(CTR) \oplus C$

Ctr overflow? Re-encrypt with a new key. Expensive!



Large storage overhead

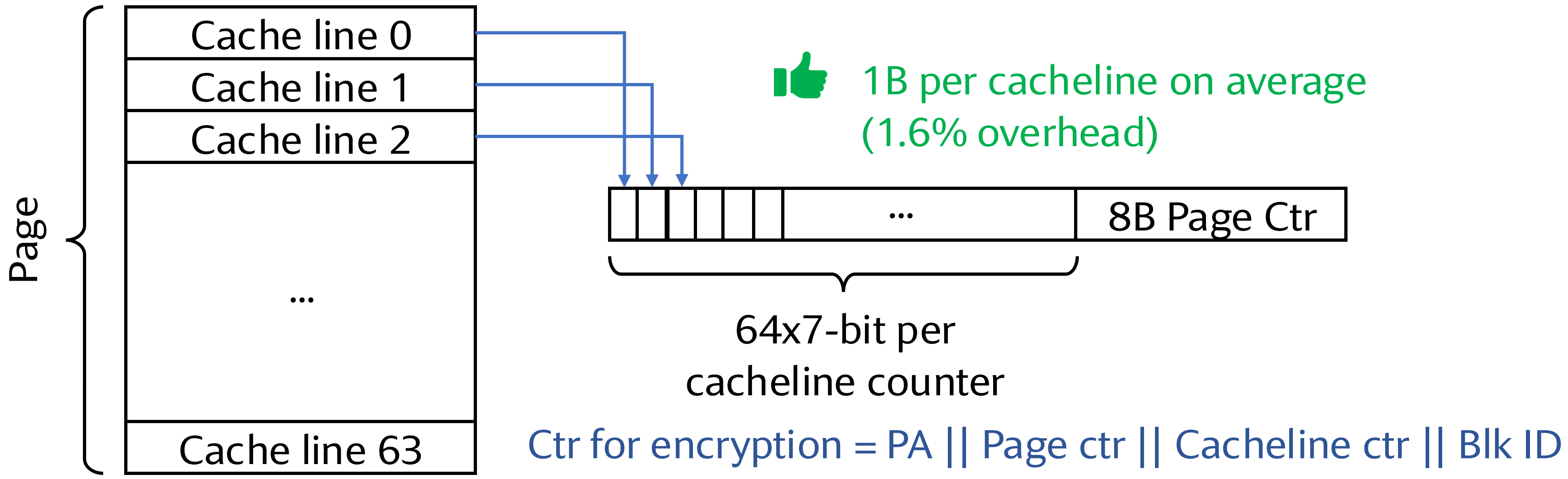
Smaller Cacheline Counter + Physical Address



Guarantees both spatial and temporal uniqueness while using a smaller counter

$C = \mathrm{Enc_k}(CTR) \oplus P \qquad P = \mathrm{Enc_k}(CTR) \oplus C$

An Even More Compact Counter Scheme



Cacheline counter overflow? Increment the page counter, reset cacheline counters, re-encrypt the entire page

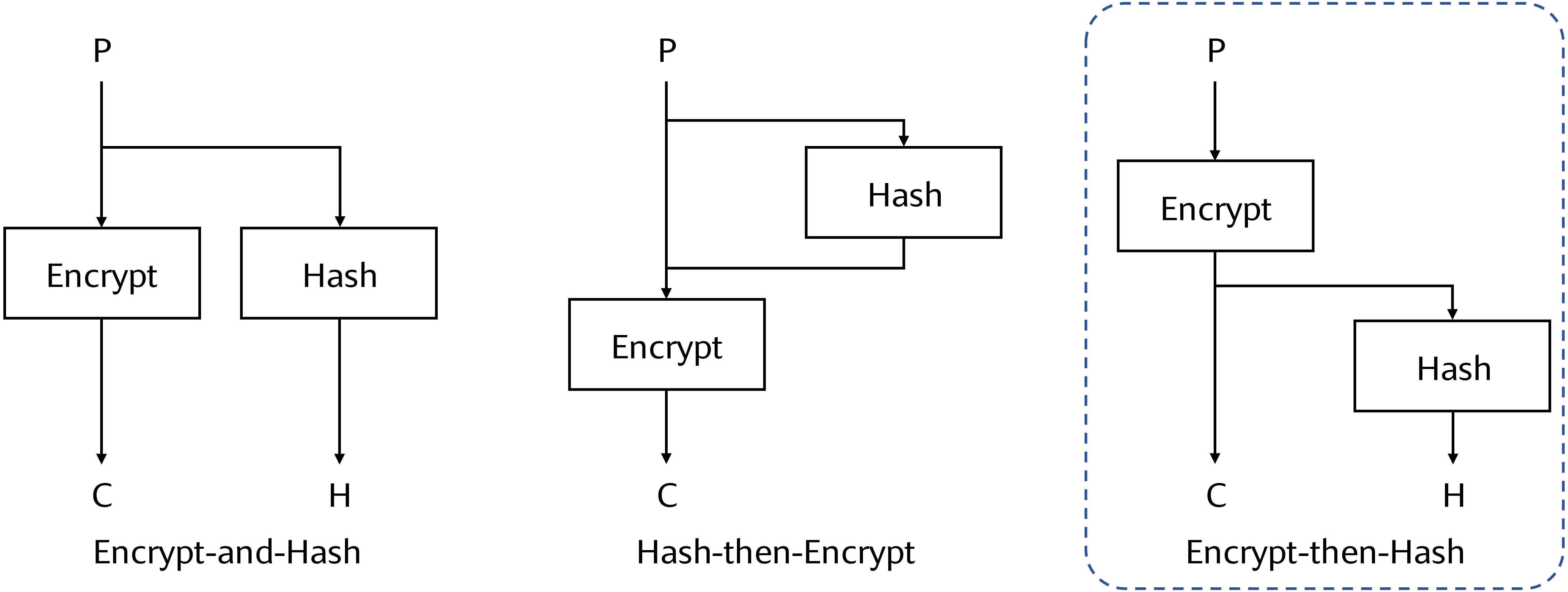
Integrity Protection Goals

- **No spoofing:** Attacker cannot replace the value
- **No splicing:** Attacker cannot exchange the value with another value from a different location
- **No replay:** Attacker cannot replay an old value from the same location

Idea 1: Use crypto hashes

Integrity Protection*

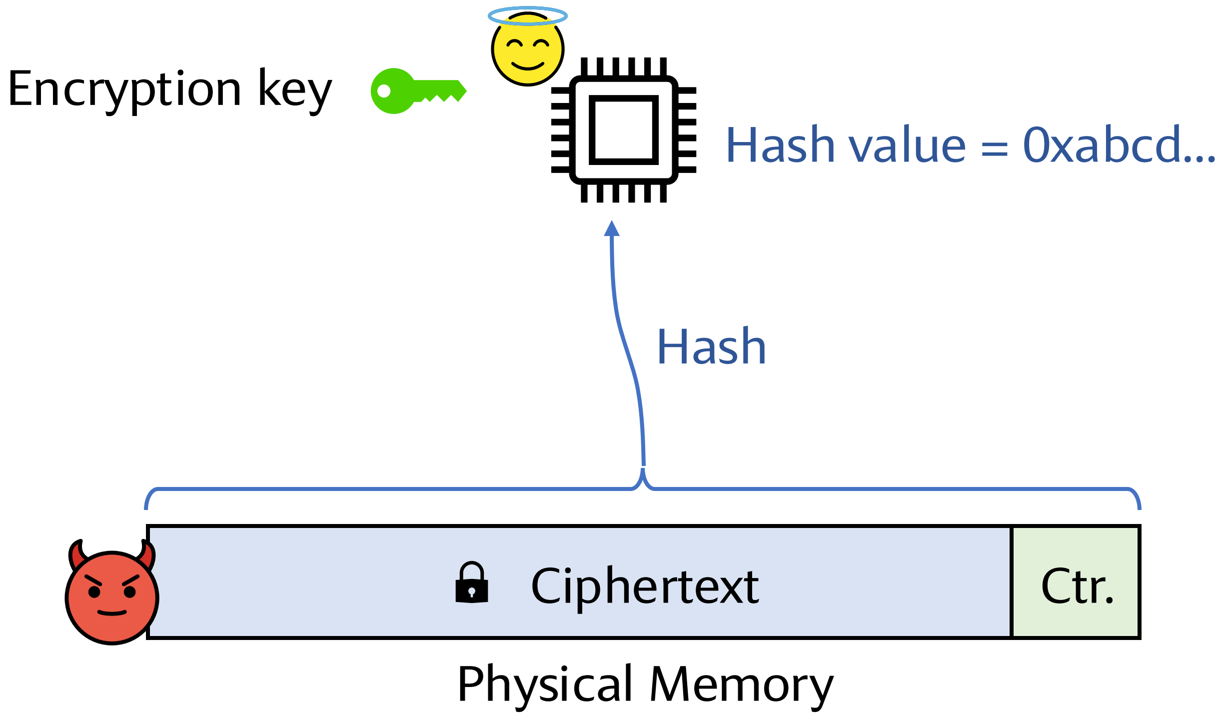
We have three schemes



*This slide is slightly wrong. We use MAC instead of Hash

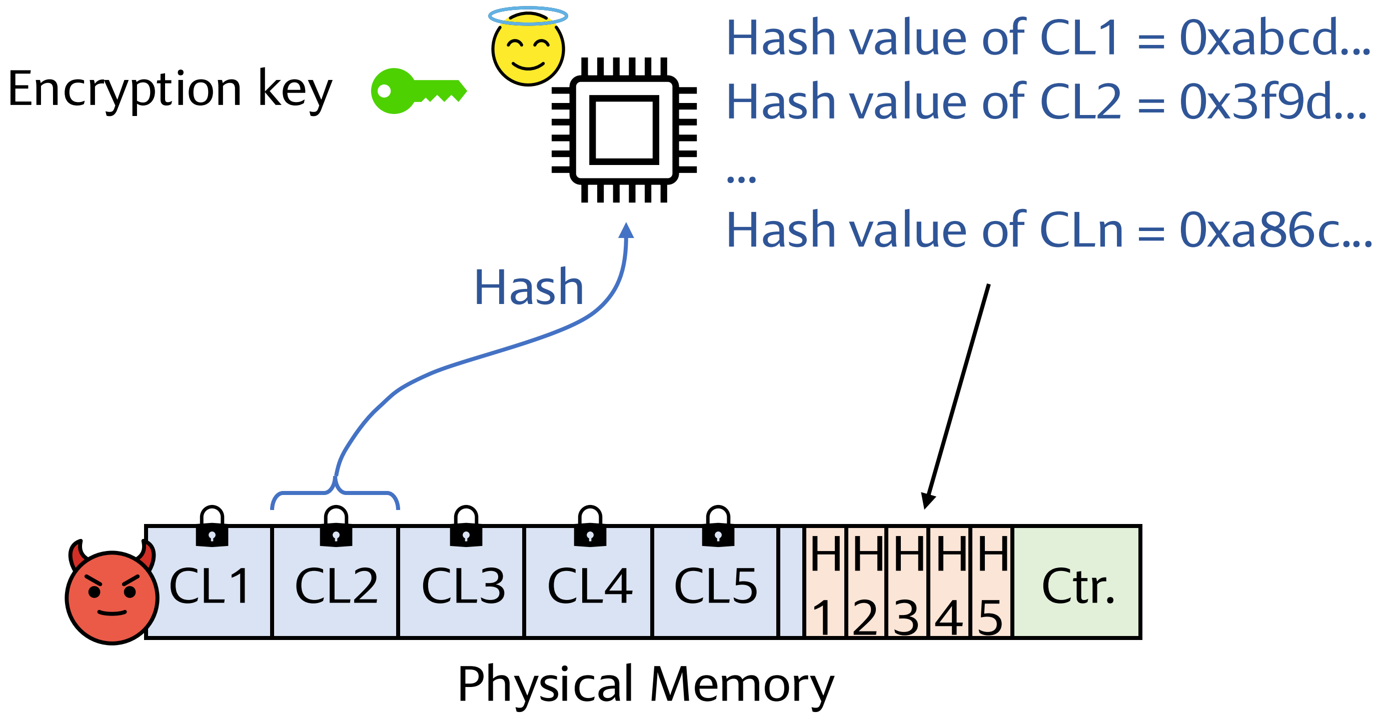
Naïve Memory Integrity Protection

Hash the entire memory and store the expected hash value on the chip



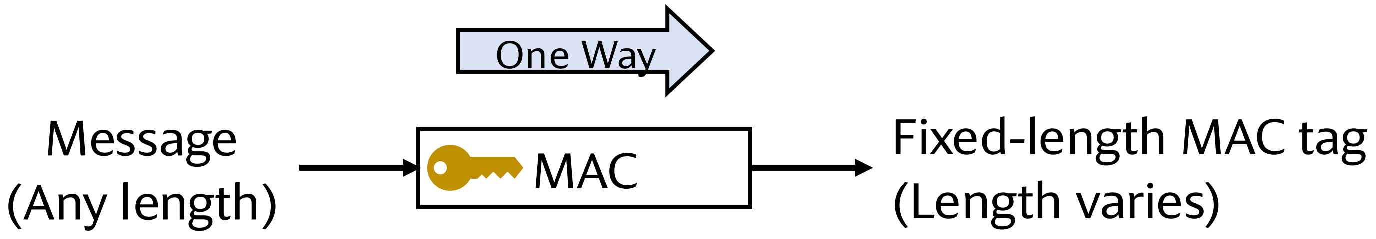
Naïve Memory Integrity Protection

Hash the entire memory and store the expected hash value on the chip



Not secure! Attacker can forge ciphertext blocks and their hashes

Hammer 5: Message Authentication Code (MAC)



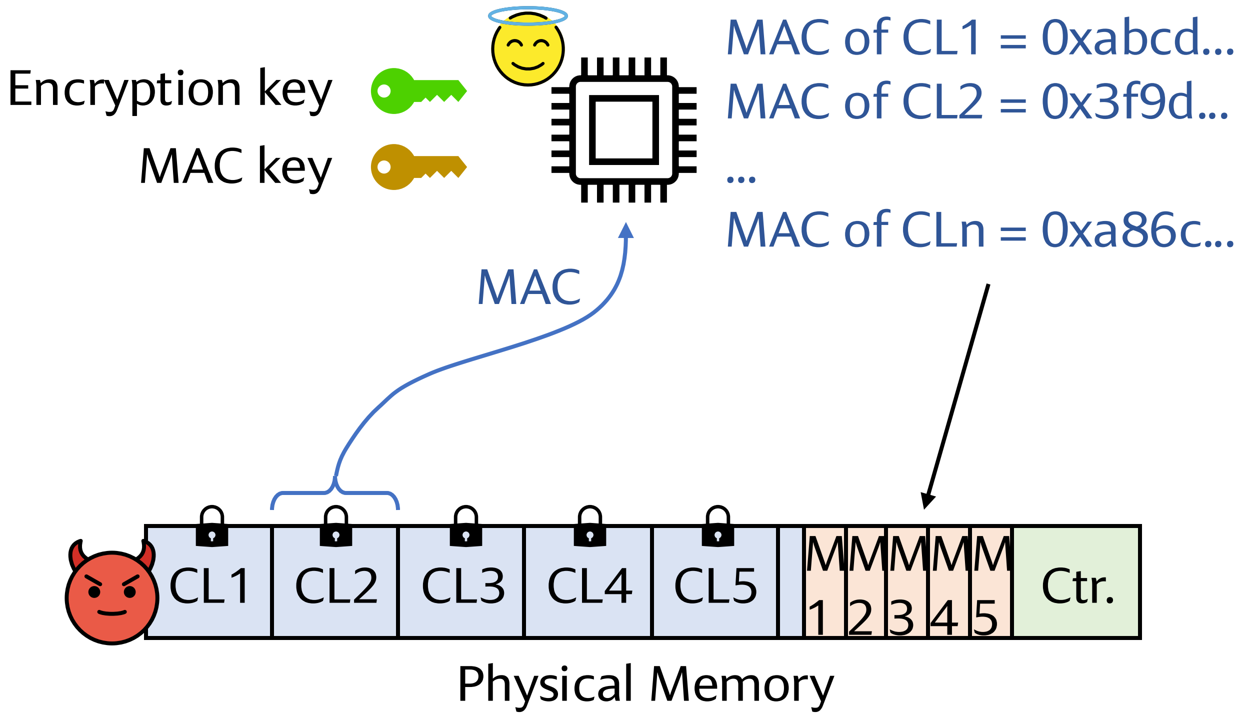
Properties:

- Verifier has the same key
- Only the person who has the key can produce the correct MAC tag
  ⇒ Correct MAC: The message is authentic

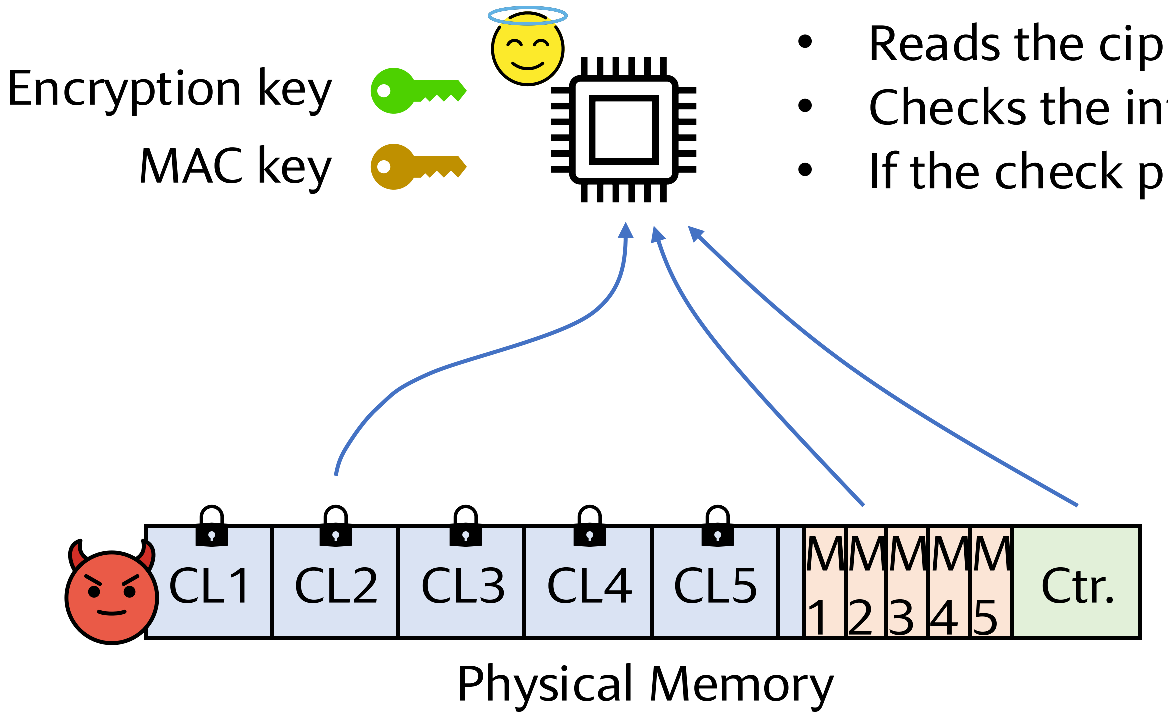
Examples:

- Hash-based MAC (HMAC): Turns a crypto hash function into a MAC construction (e.g., HMAC-SHA256)
- Poly1305: A dedicated MAC design by DJB. Commonly used with ChaCha20, a stream cipher

Memory Integrity Protection with MAC



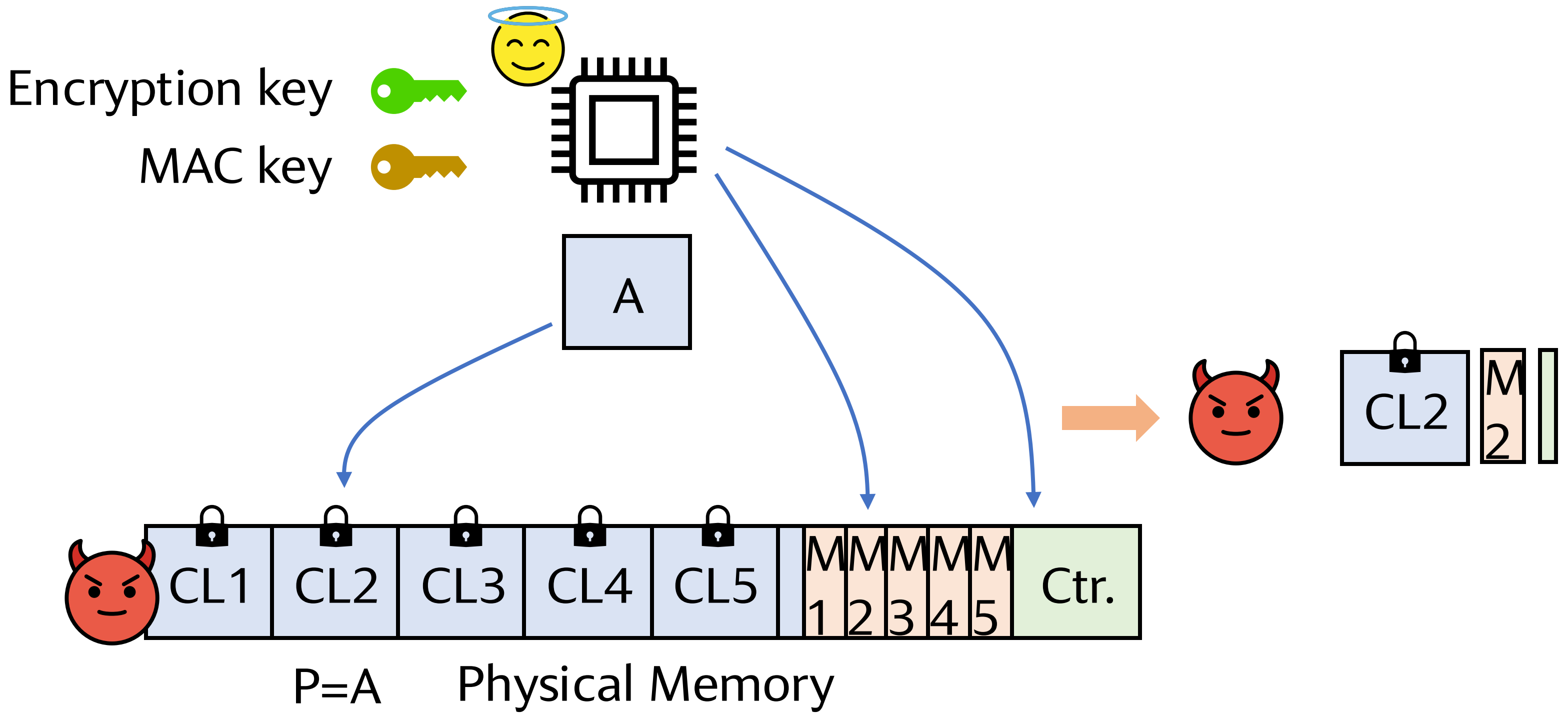
Accessing the Memory



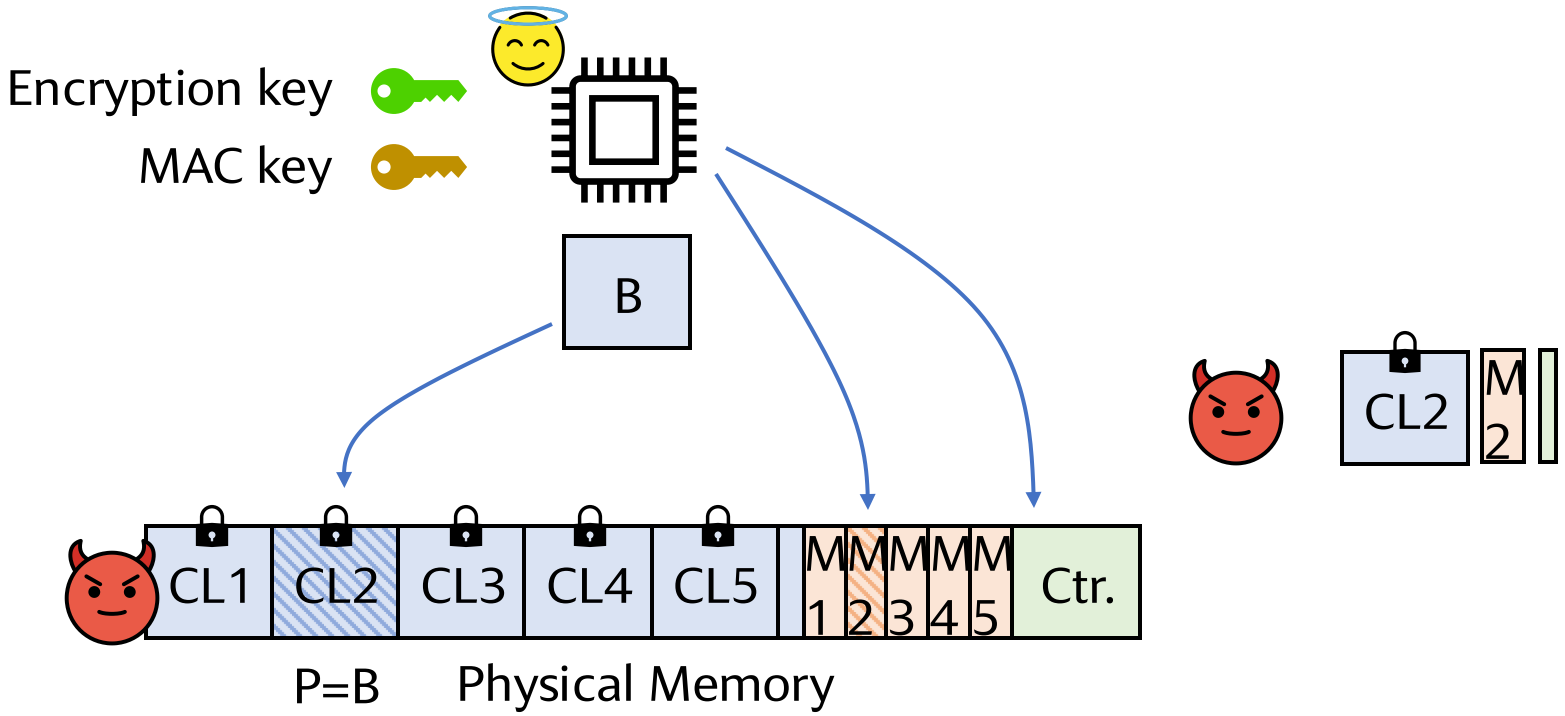
- Reads the ciphertext, counter, and MAC
- Checks the integrity of the ciphertext
- If the check passes, decrypts the ciphertext

To prevent splicing: MAC Tag = MAC_k(Ciphertext, PA)

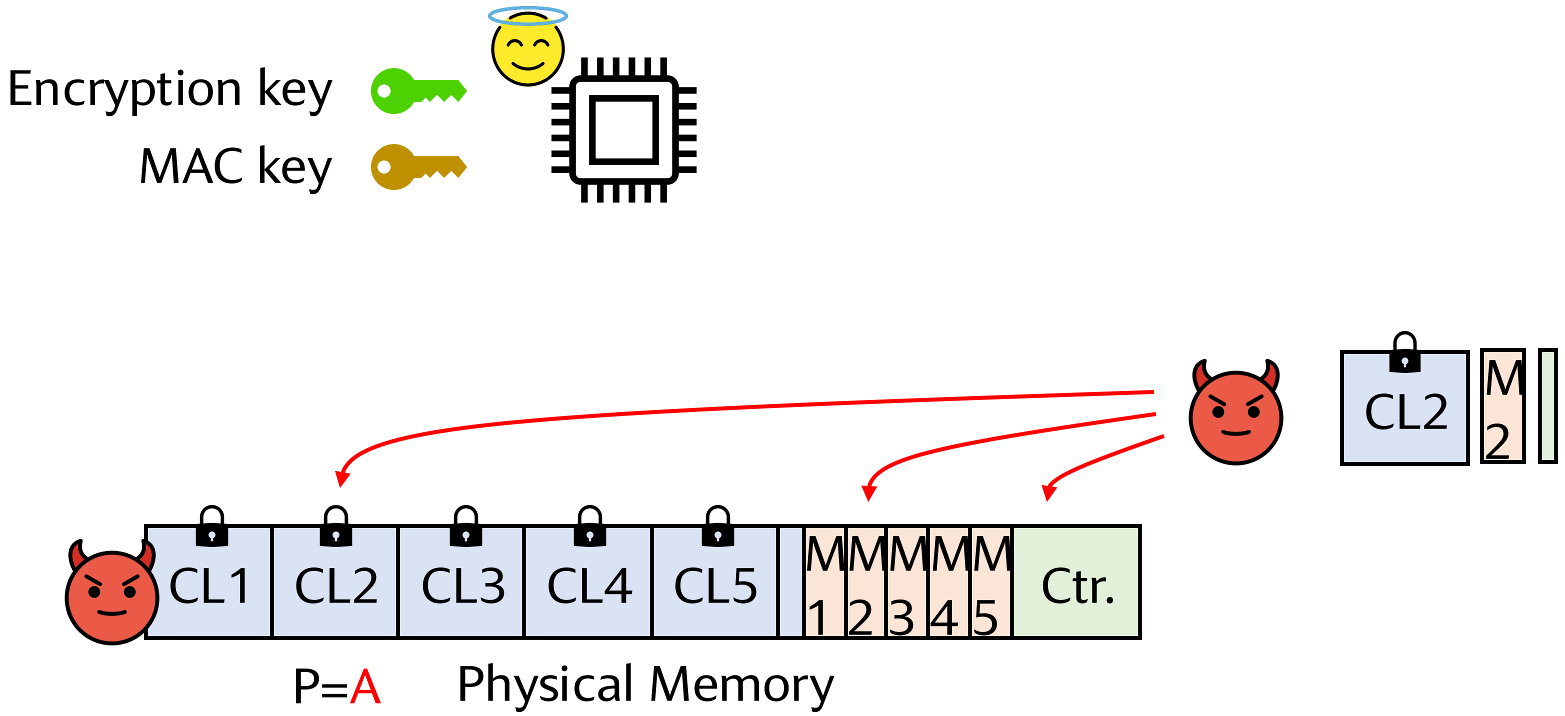
Wait, What About Freshness



Wait, What About Freshness

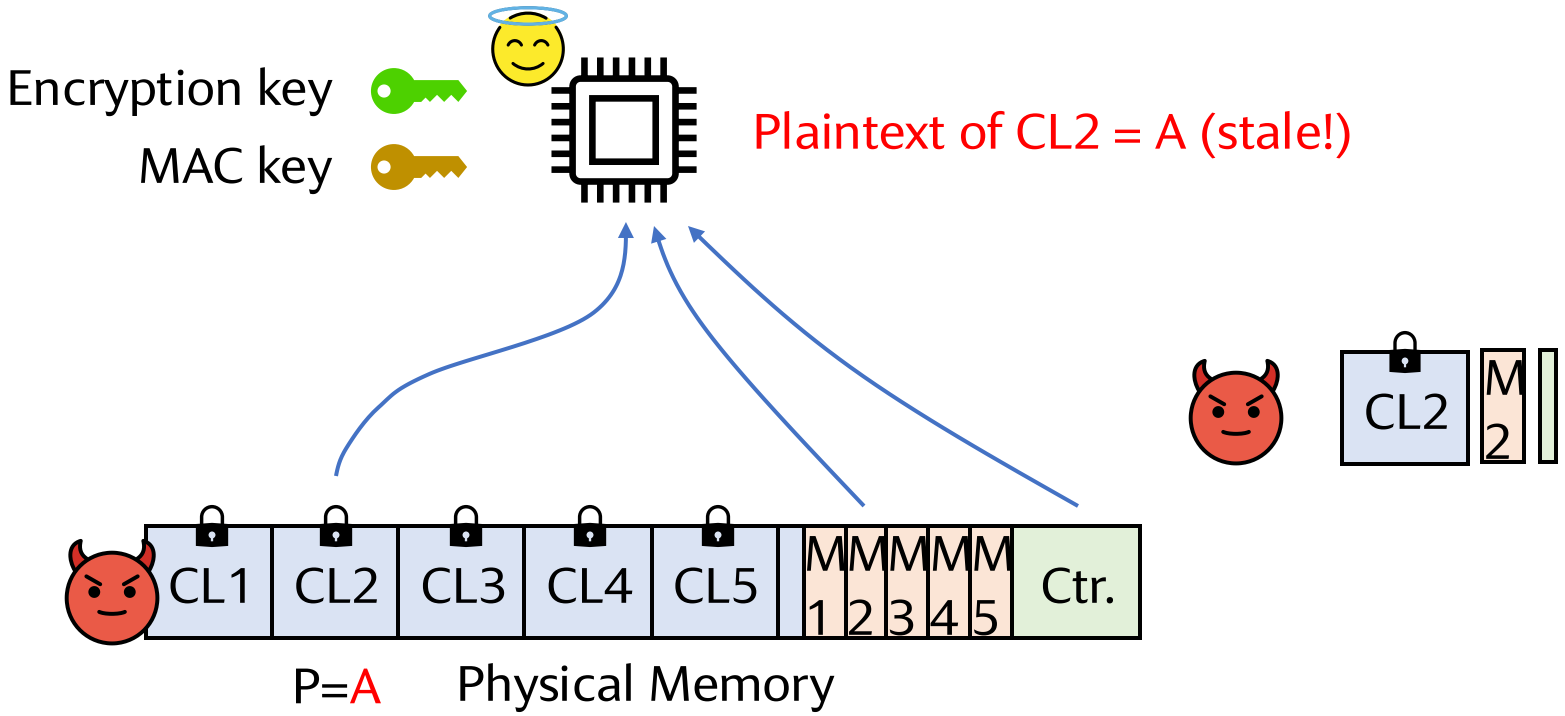


Wait, What About Freshness

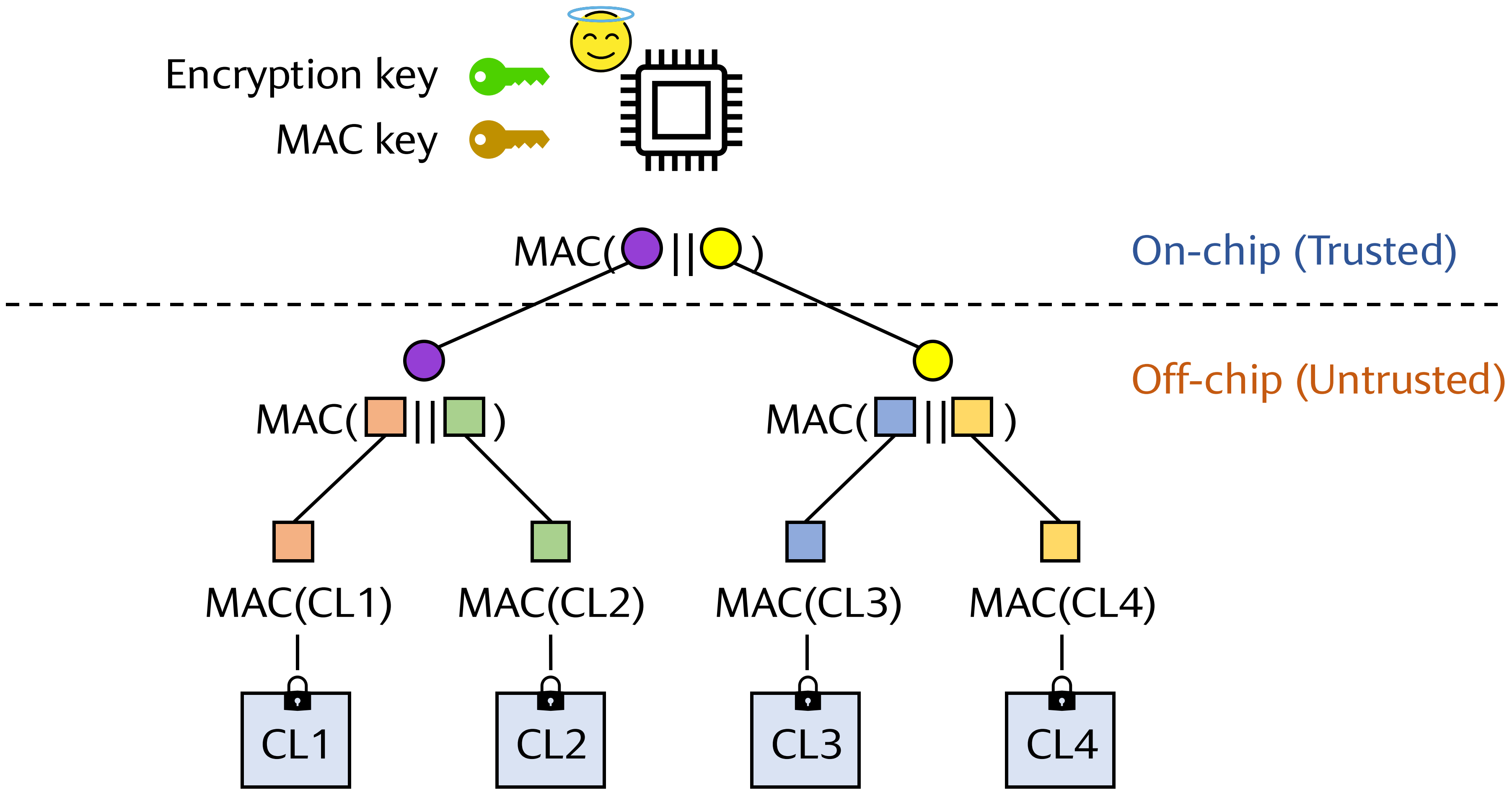


Wait, What About Freshness

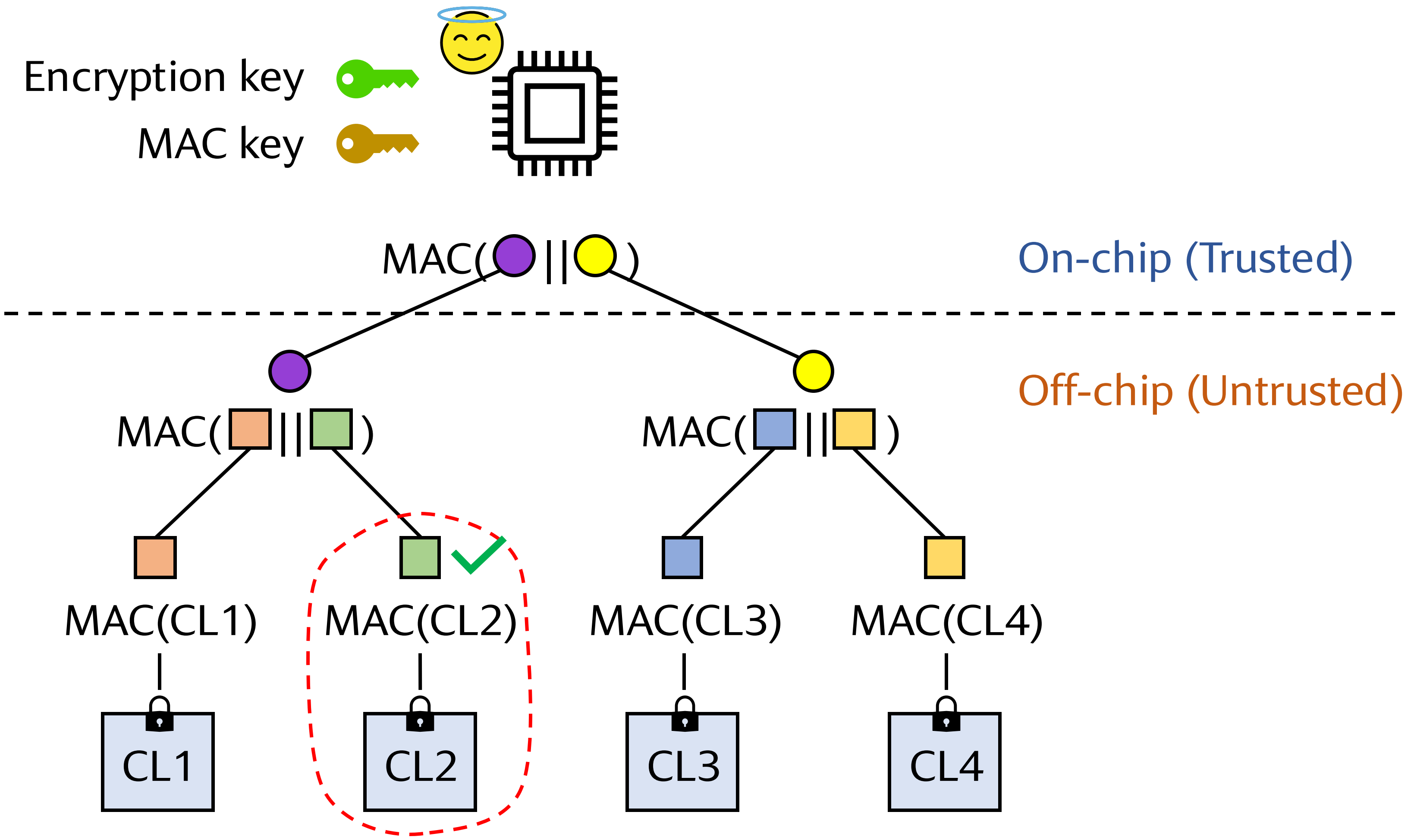
Authenticating a cache line in isolation isn't enough! We need a MAC that covers the entire memory. How to do that efficiently?



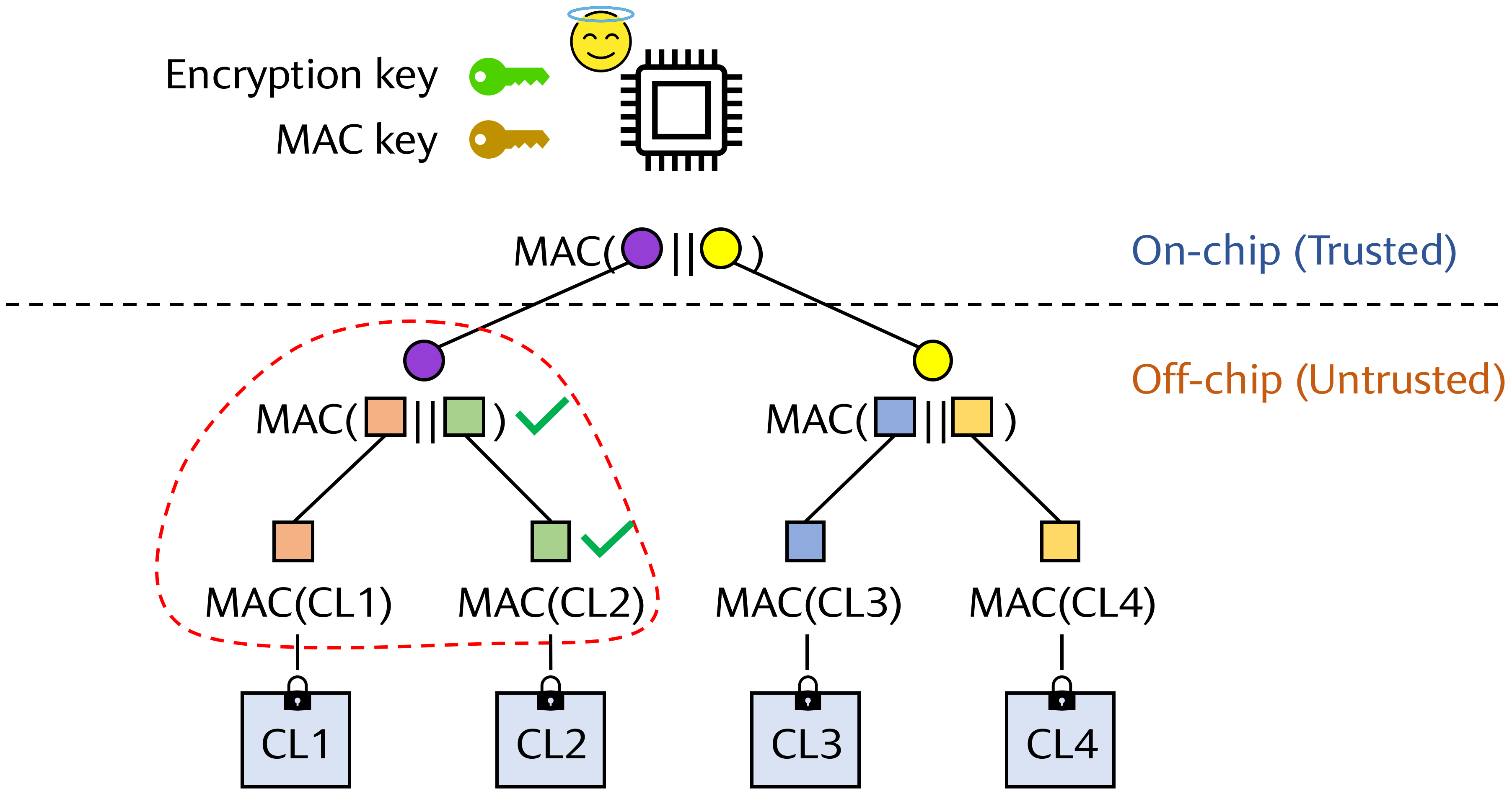
Hammer 6: Merkle (Hash) Tree



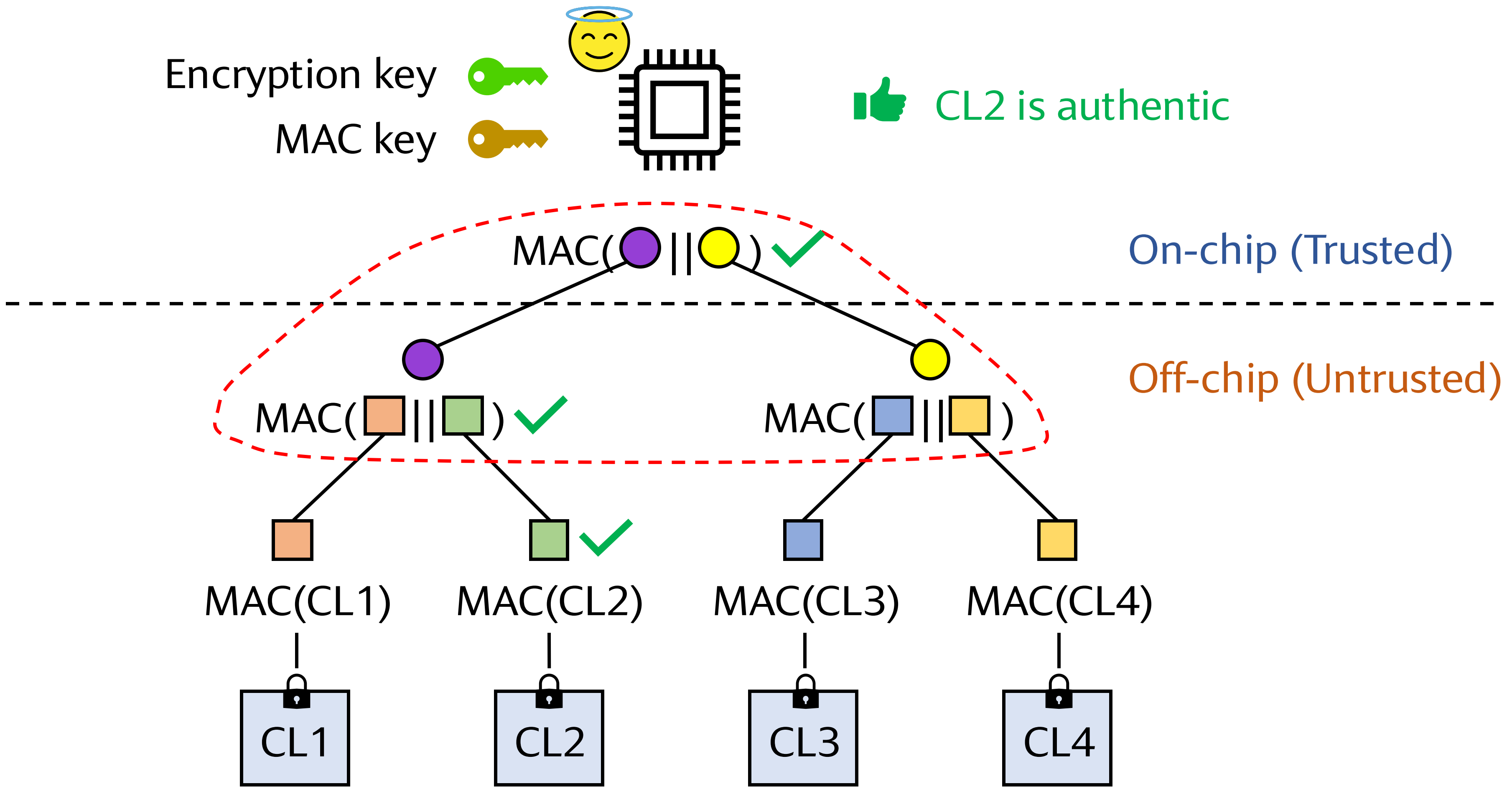
Verify the Integrity of C2



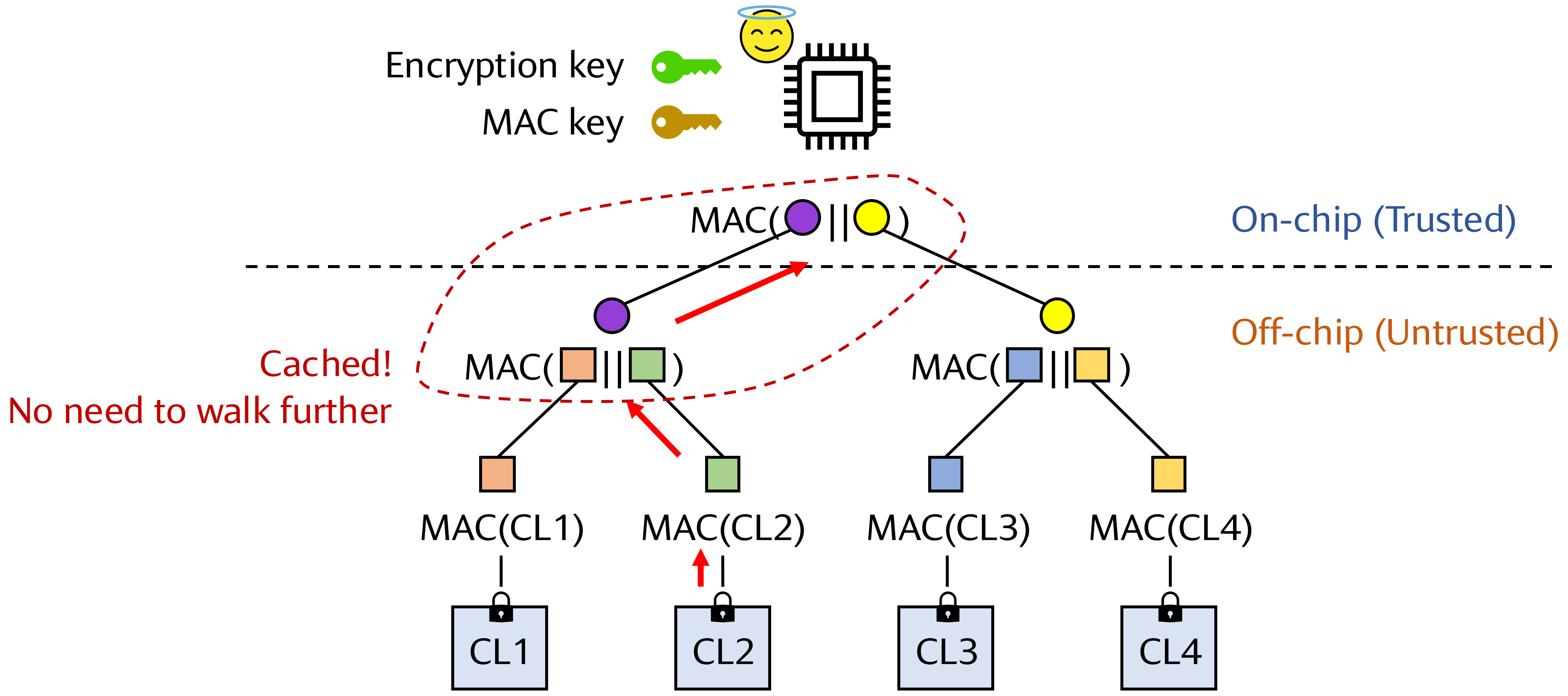
Verify the Integrity of C2



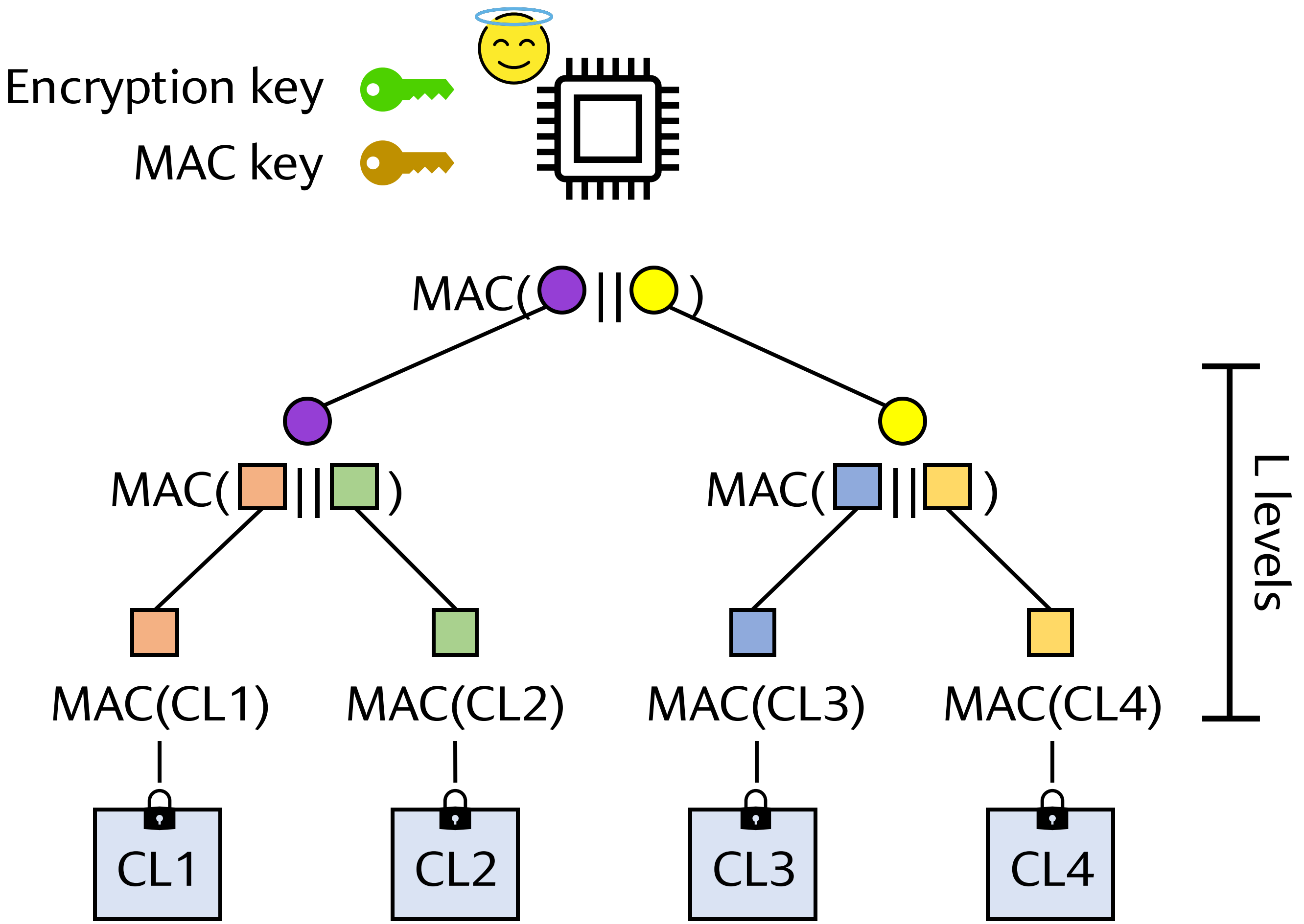
Verify the Integrity of CL2



Verify the Integrity of CL2



How Large is the Merkle Tree



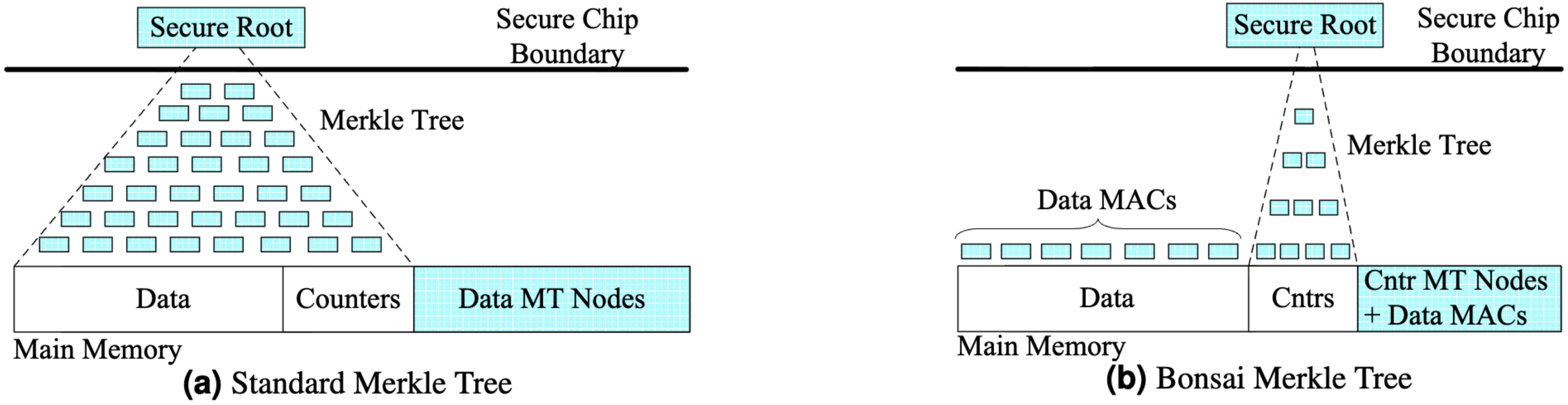
Assume 64-bit (or 8-B) MAC
⇒ 8 MACs per cacheline ⇒ Fan-out = 8

Total leaves # $N_{leaf} = 8^L$
Total tree nodes # $N_{node} = (8^{L+1} - 8)/7$

cachelines covered = N_{leaf}
Absolute storage cost = $8N_{node}$
Storage overhead =

$$\frac{8N_{node}}{64N_{leaf}} = \frac{8^{L+1} - 8}{7 \times 8^{L+1}} = \frac{1 - 8^{-L}}{7} \approx \frac{1}{7}$$

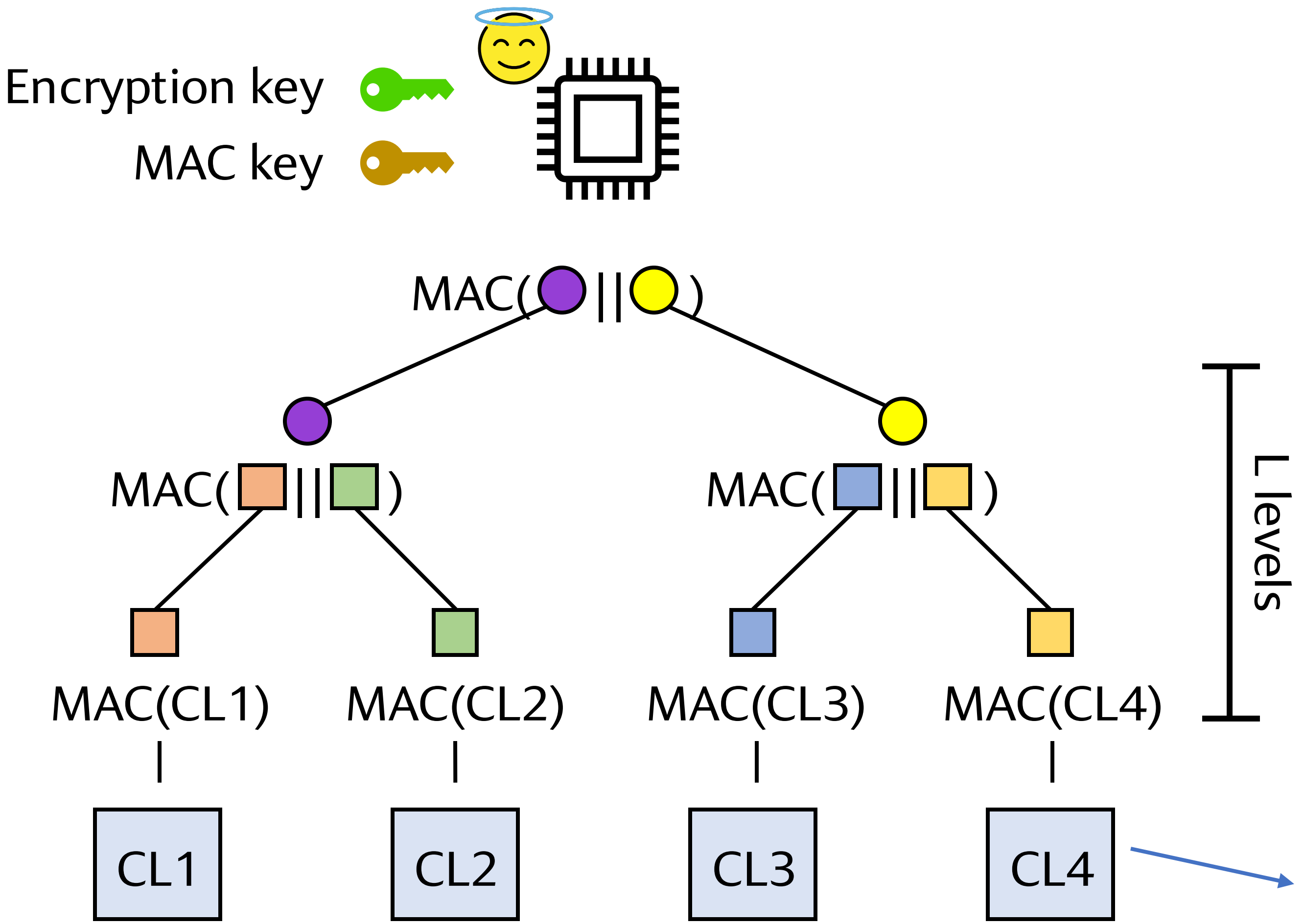
Bonsai Merkle Tree (BMT)*



To use BMT, the Data MAC needs to cover (1) the ciphertext, (2) PA, and (3) Counter
i.e., MAC Tag = MAC_k(Ciphertext, PA, Ctr)

*Rogers et al. "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly" (MICRO '07)

How Large is the Merkle Tree



Assume 64-bit (or 8-B) MAC
⇒ 8 MACs per cacheline ⇒ Fan-out = 8

Total leaves # $N_{leaf} = 8^L$
Total tree nodes # $N_{node} = (8^{L+1} - 8)/7$

cachelines covered = $64N_{leaf}$
Absolute storage cost = $8N_{node}$
Storage overhead =

$$\frac{8N_{node}}{64 \times 64N_{leaf}} = \frac{1 - 8^{-L}}{64 \times 7} \approx \frac{1}{448}$$

Counters for 64 cachelines

Overall Storage Overhead