Real World Cryptography Conference 2016
6-8 January 2016, Stanford, CA, USA

Intel® Software Guard Extensions (Intel® SGX) **Memory Encryption Engine (MEE)**

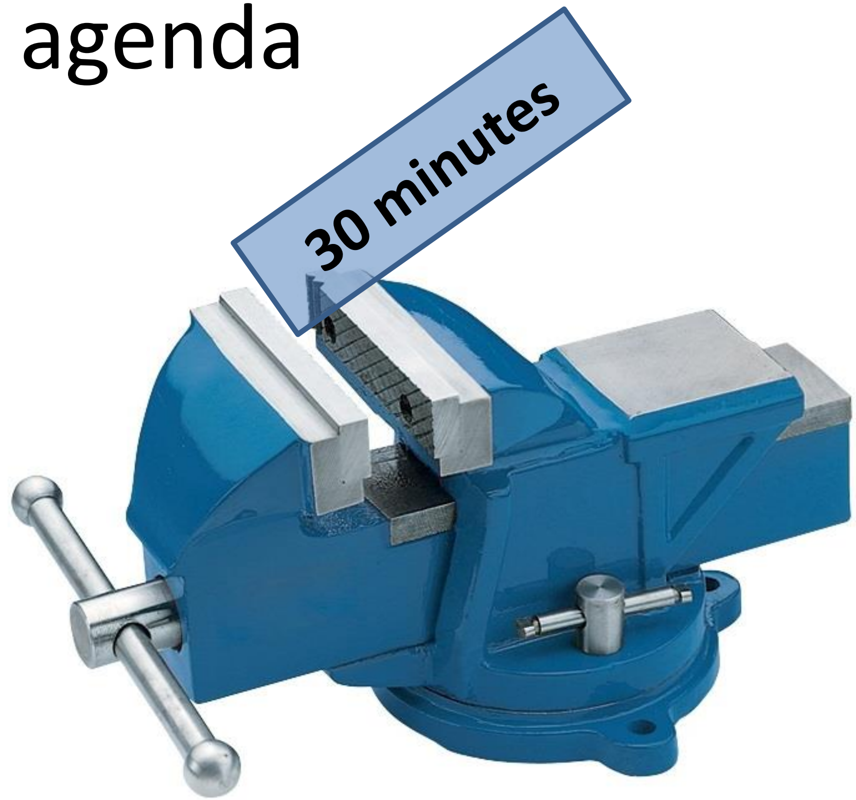
Shay Gueron

Intel Corp., Intel Development Center, Haifa, Israel

University of Haifa, Israel

(real world) agenda

- Describe in a nutshell
  - Why Memory Encryption
    - Some real world challenges
  - How it was done
    - Real world considerations
  - Security bounds
    - Real world security bounds
  - Performance
    - Real world performance experiment



Cryptographic protection of memory

- An **essential ingredient** for any technology that allows a closed computing system to
- **Run software in a trustworthy manner and to handle secrets**
- **While external memory susceptible to snooping & tampering**

- Example: **Intel® Software Guard Extensions (Intel® SGX)**
  - 6^{th} Generation Intel® Core™ (Architecture codename Skylake)
  - The assumed security perimeter includes only the CPU package internals **DRAM is untrusted**.

SGX cryptographic protection of memory is supported by the Memory Encryption Engine

Memory **E**ncryption **E**ngine

- Hardware unit - extension of the Memory Controller
- **Objectives:**
  - **Data Confidentiality**: Collections of memory images of DATA written to the DRAM (into different addresses and points in time) cannot be distinguished from random data.
  - **Integrity**: DATA read back from DRAM to LLC is the same DATA that was most recently written from LLC to DRAM.

- MEE is **not** an Oblivious RAM
  - Does not hide the fact that data is written to the DRAM, when it is written, and to which physical address

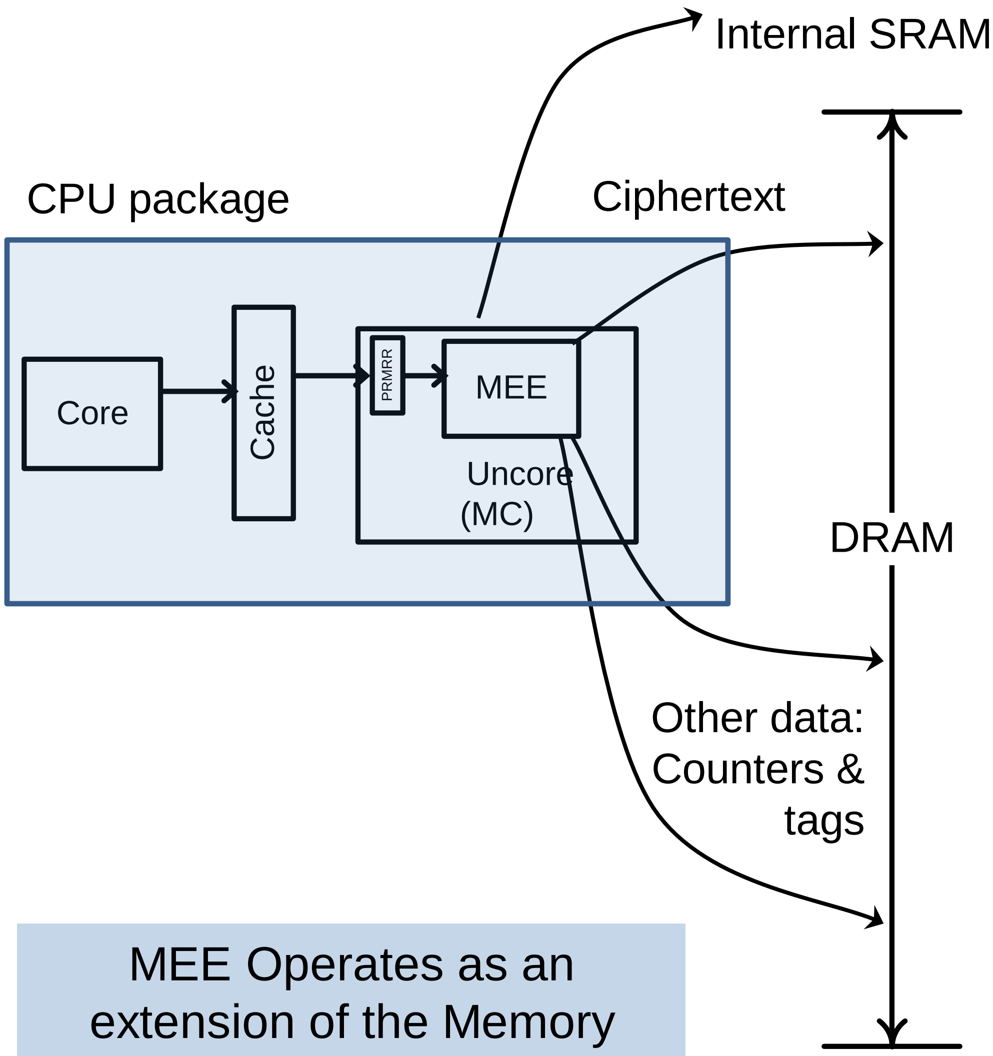
Memory **E**ncryption **E**ngine
Real World Challenge

- The challenge: adding a hardware unit to the micro architecture of a **general purpose processor** **(real product)**
- **Requires design under very strict engineering constraints**
  - Minimal hardware area **but** tolerable performance
  - A small budget for internal storage
  - Standard crypto primitives are not optimal for this problem
  - Since transparent encryption is not enough
    - MEE needs to **initiate additional** memory transactions

How the MEE works – in a nutshell

Build

- Core issues a transaction
  - (to MEE region); e.g., WRITE
- Transaction misses caches and forwarded to **M**emory **C**ontroller
- MC detects address belongs to MEE region & routes transaction to MEE
- Crypto processing and… …
- MEE **initiates additional memory accesses** to obtain (or write to) necessary data from DRAM
  - Produces plaintext (ciphertext)
  - Computes authentication tags
  - (uses/updates internal data)
  - writes ciphertext + added data



MEE Operates as an extension of the Memory Controller (MC)

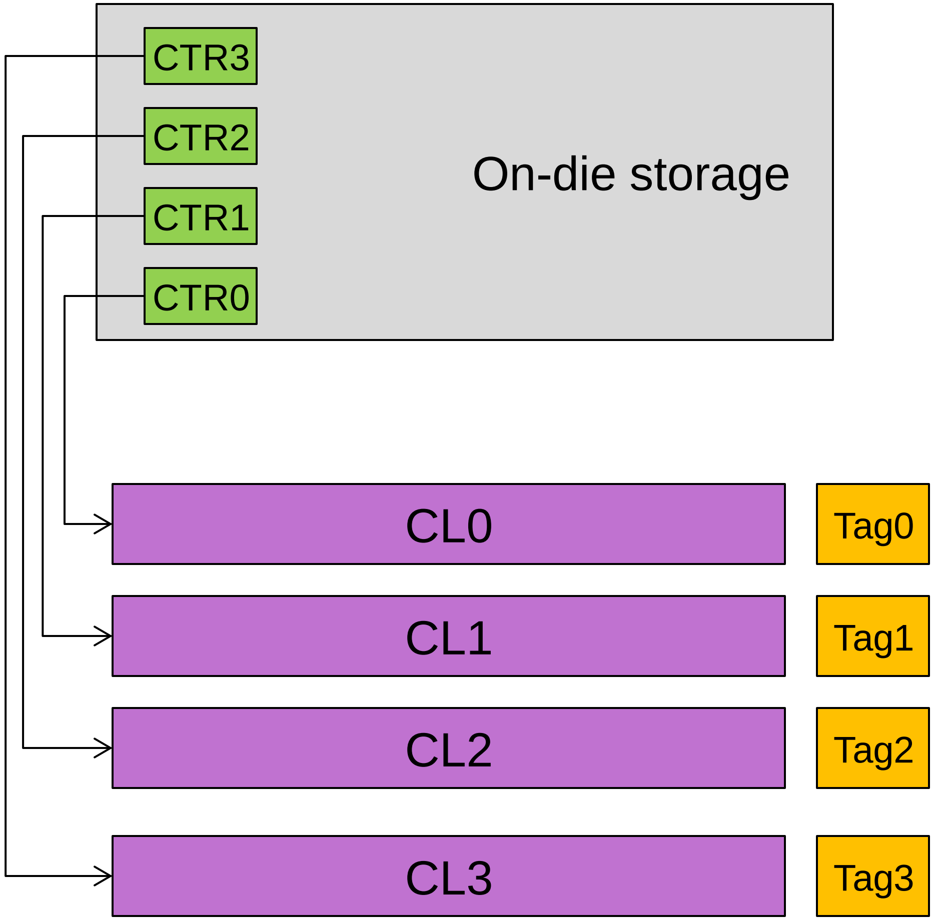
MEE basic setup and policy

- Memory access always at 512 bits **C**ache **L**ine (**CL**) granularity
- Keys: randomly generated at reset by a HW DRNG module
  - Accessible only to MEE hardware

- Drop-and-lock policy: upon MAC tag mismatch, MEE
    - **Drops** the transaction (i.e., **no data is sent to the LLC**)
    - **Locks** the MC (i.e., **no further transactions are serviced**).
    - Eventually system **halts & reset is required** (with new keys)

- **No unauthenticated data ever infiltrate the CPU boundary**
  - **While internal calculations can be parallelized at any order**
- **Adversary has only one failed forgery attempt per key**

An abstract 1-level data structure

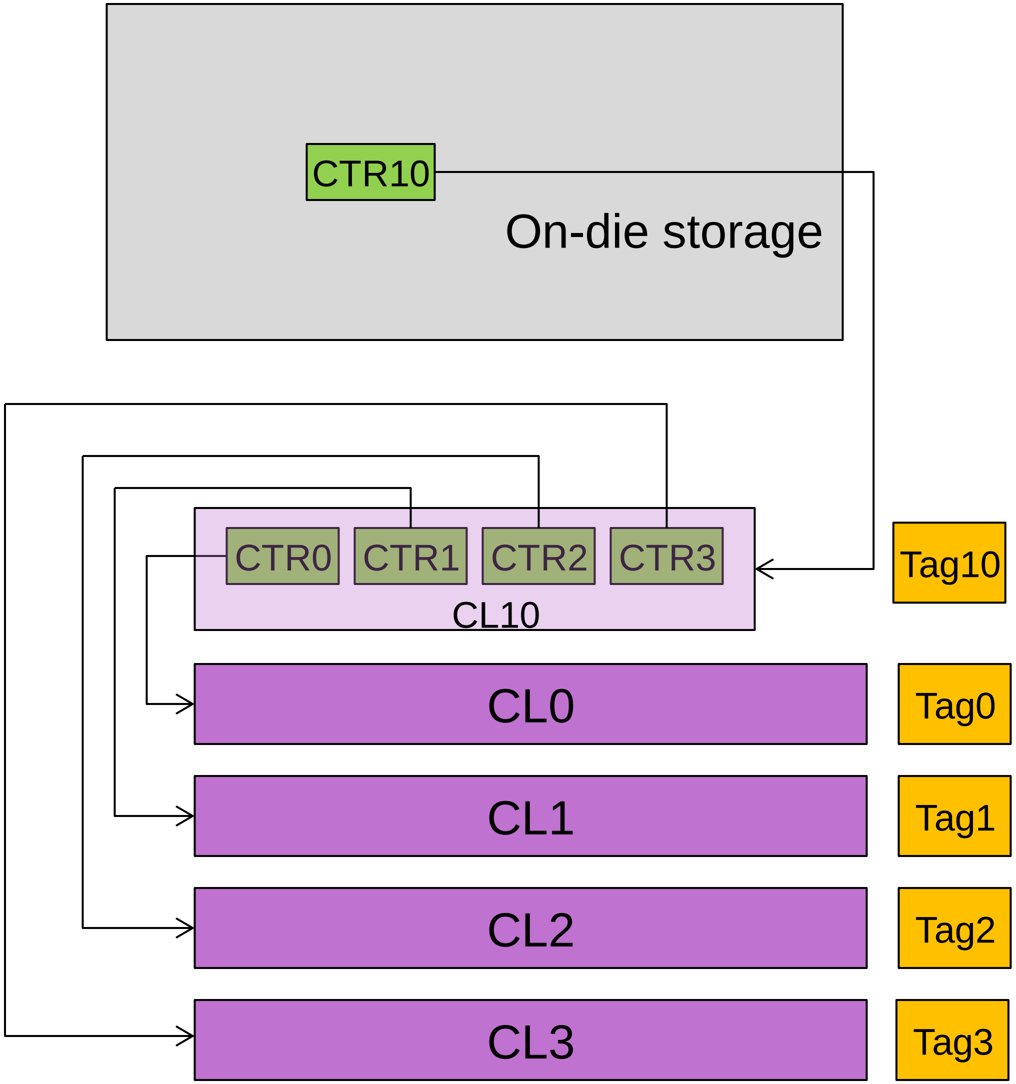


- A "**Stateful**" MAC algorithm over Data + CTR
- (internal) CTR's are trusted

√ Integrity + replay protection

- **Constraint:**
  - **Internal storage (SRAM) is very expensive**

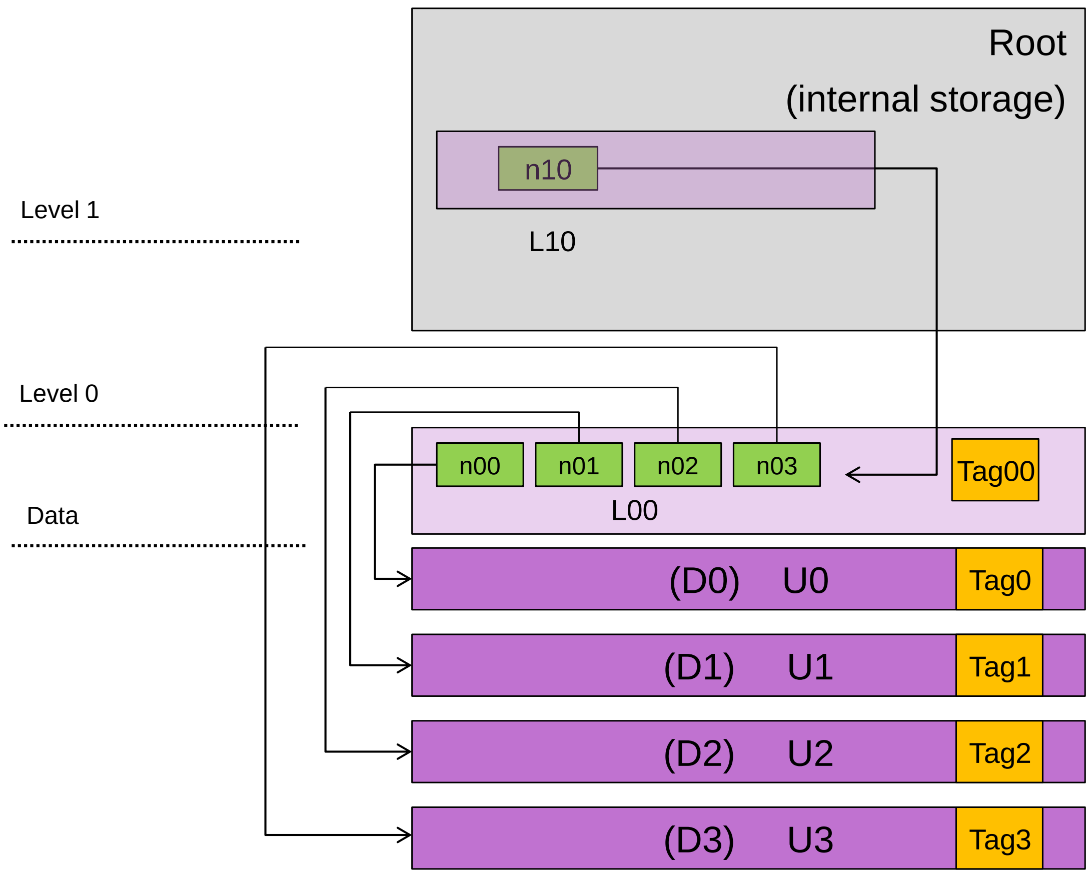
Compressing it: a 2-level data structure



- "**Stateful**" MAC over Data and CTR
- 1st level tags protect Data
- 2nd level tag protects the counters
- Top level tag is internal → trusted
- Counters protect "freshness"

- **Trading internal storage with a walk over the data structure**
- **(complexity & performance)**

Embedded MAC tags



Memory accesses can be saved if tags are **embedded** in the CL's

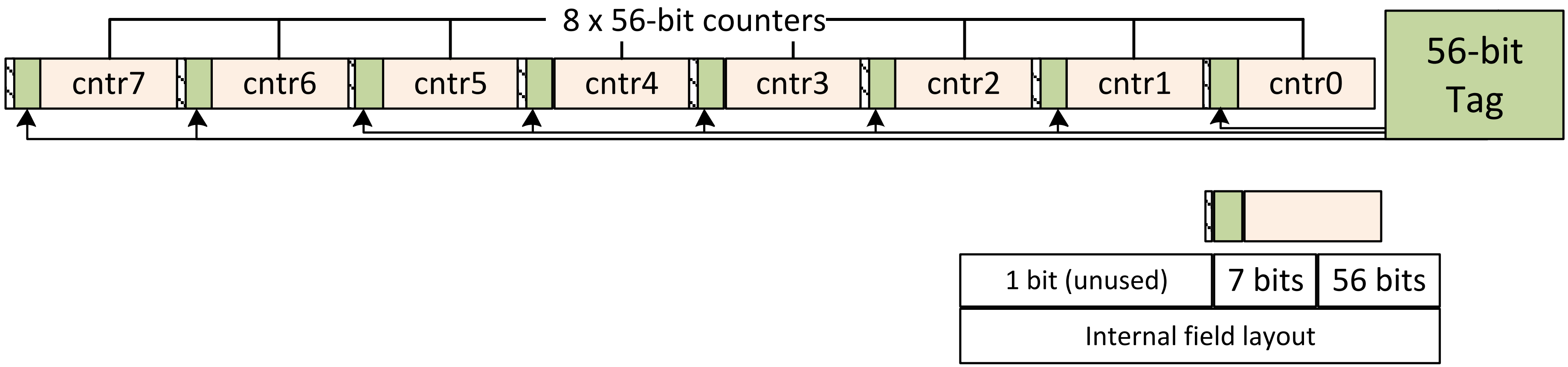
Possible in case some bits in the CL can be reserved for the tags

Idea:

$56 \times 8 + 64 = 512$

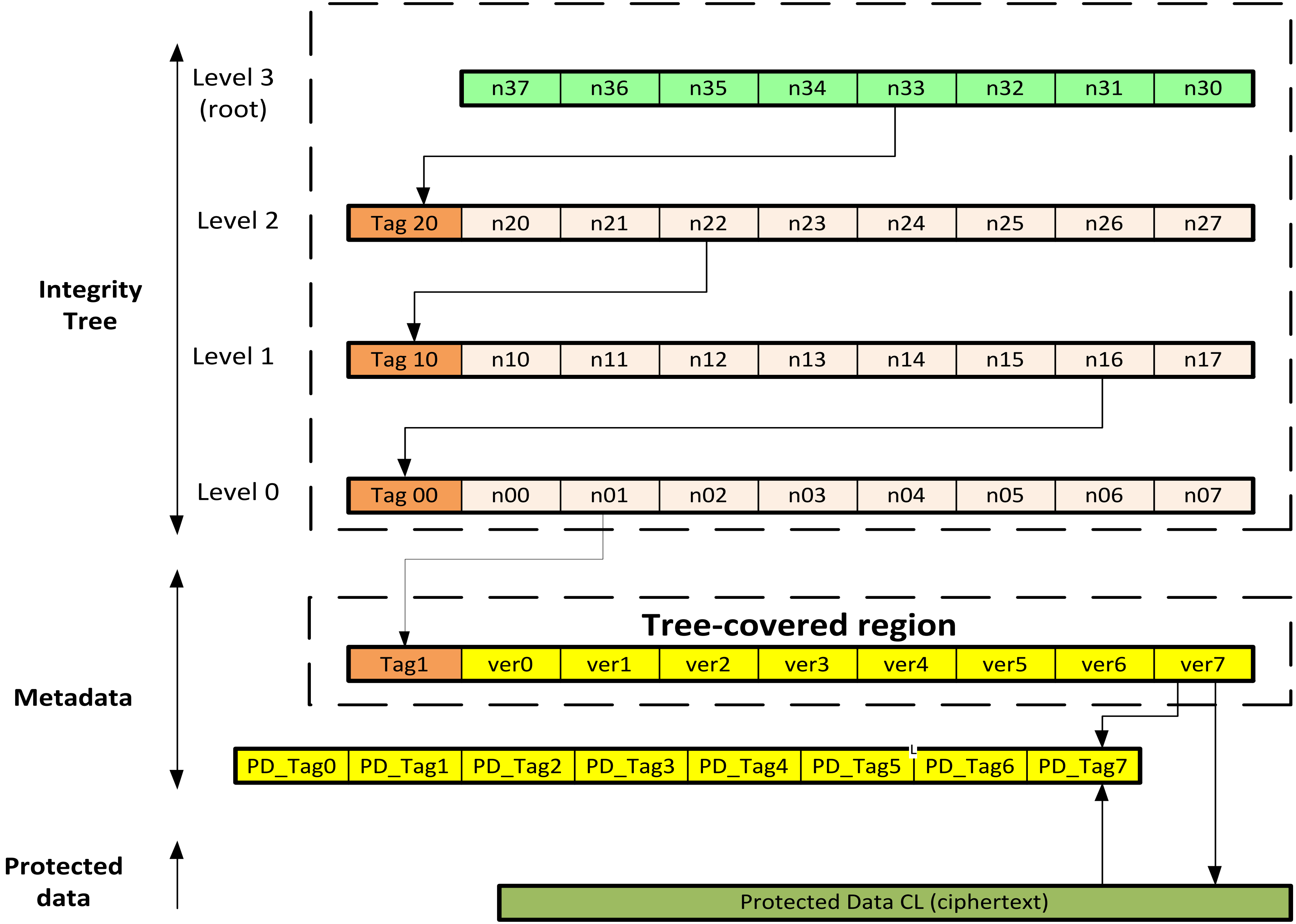
Embedded MAC tags

The MEE inequality **56 × 8 + 56 < 512**

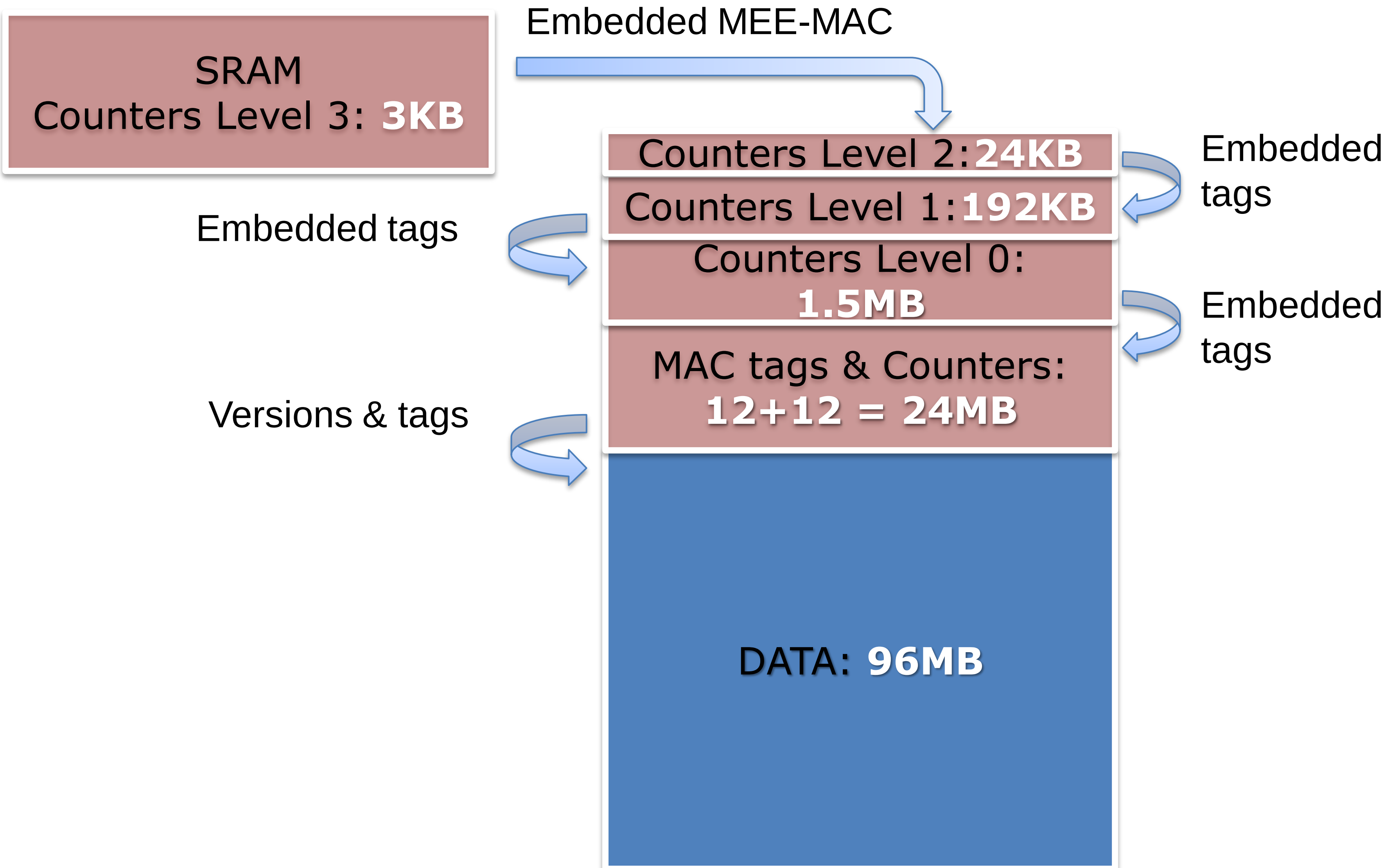


One CL accommodates 8 counters and **embedded tag**

The MEE actual integrity tree
is a multi-level construction
with 8x compression ratio per level


Level 3
(root)
n37
n36
n35
n34
n33
n32
n31
n30
Level 2
Tag 20
n20
n21
n22
n23
n24
n25
n26
n27
Integrity
Tree
Level 1
Tag 10
n10
n11
n12
n13
n14
n15
n16
n17
Level 0
Tag 00
n00
n01
n02
n03
n04
n05
n06
n07
Tree-covered region
Tag1
ver0
ver1
ver2
ver3
ver4
ver5
ver6
ver7
Metadata
PD_Tag0
PD_Tag1
PD_Tag2
PD_Tag3
PD_Tag4
PD_Tag5
PD_Tag6
PD_Tag7
Protected
data
Protected Data CL (ciphertext)

The overall compression rate



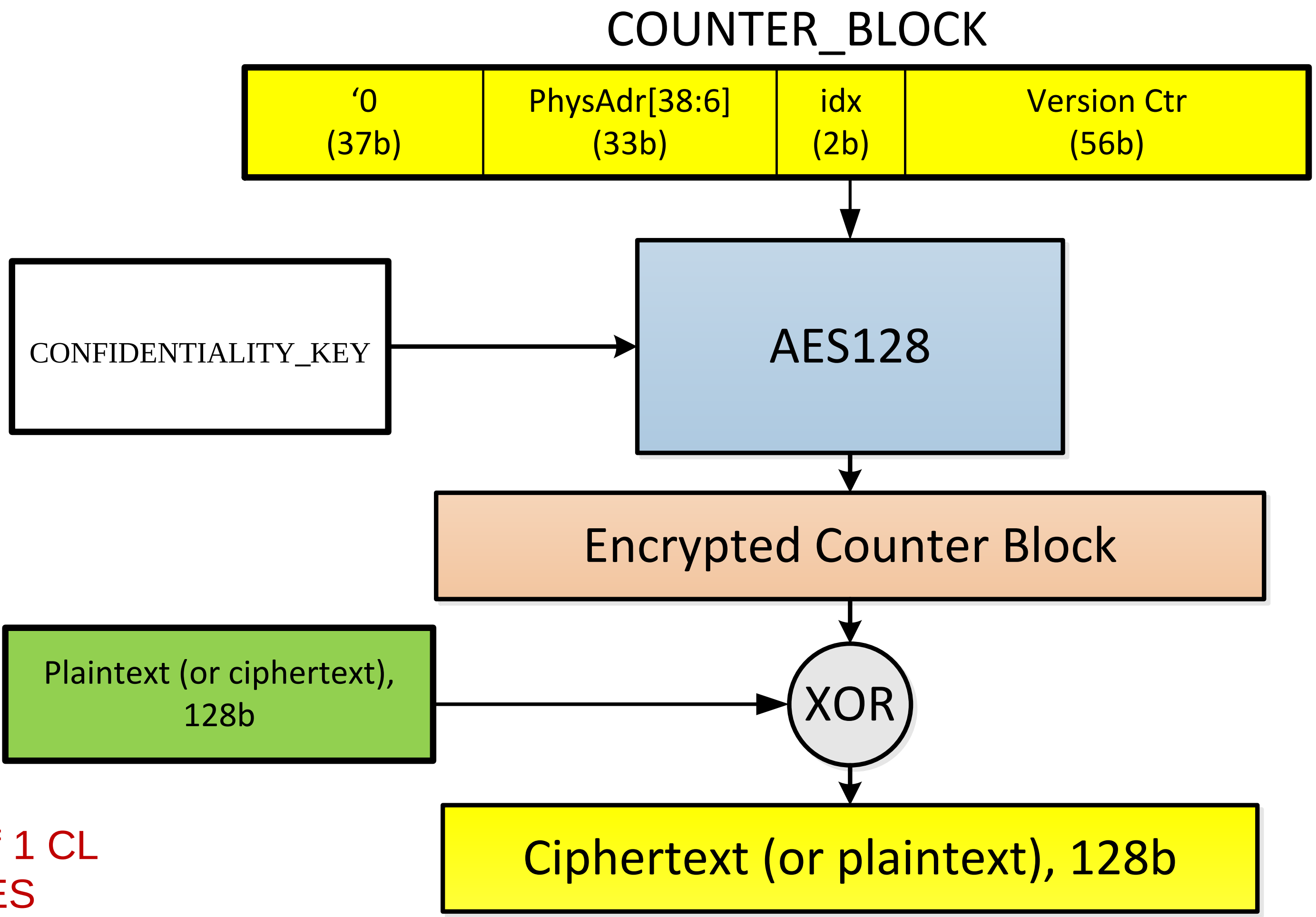
The MEE cryptographic primitives

- **A tailored AES CTR encryption**
  - Spatial and temporal “coordinates”
- **A tailored MAC algorithm**
  - Carter-Wegman MAC
    - over a multilinear universal hash function
  - Plus truncation (to 56 bits)
  - Spatial and temporal “coordinates”
- **MEE keys** (768 bits)
  - Confidentiality key: 128 bits
  - Integrity keys: Masking key: 128 bits + hash key: 512 bits

MEE Counter Mode

Spatial and temporal coordinates
identify every 16B block in the address space, at any time

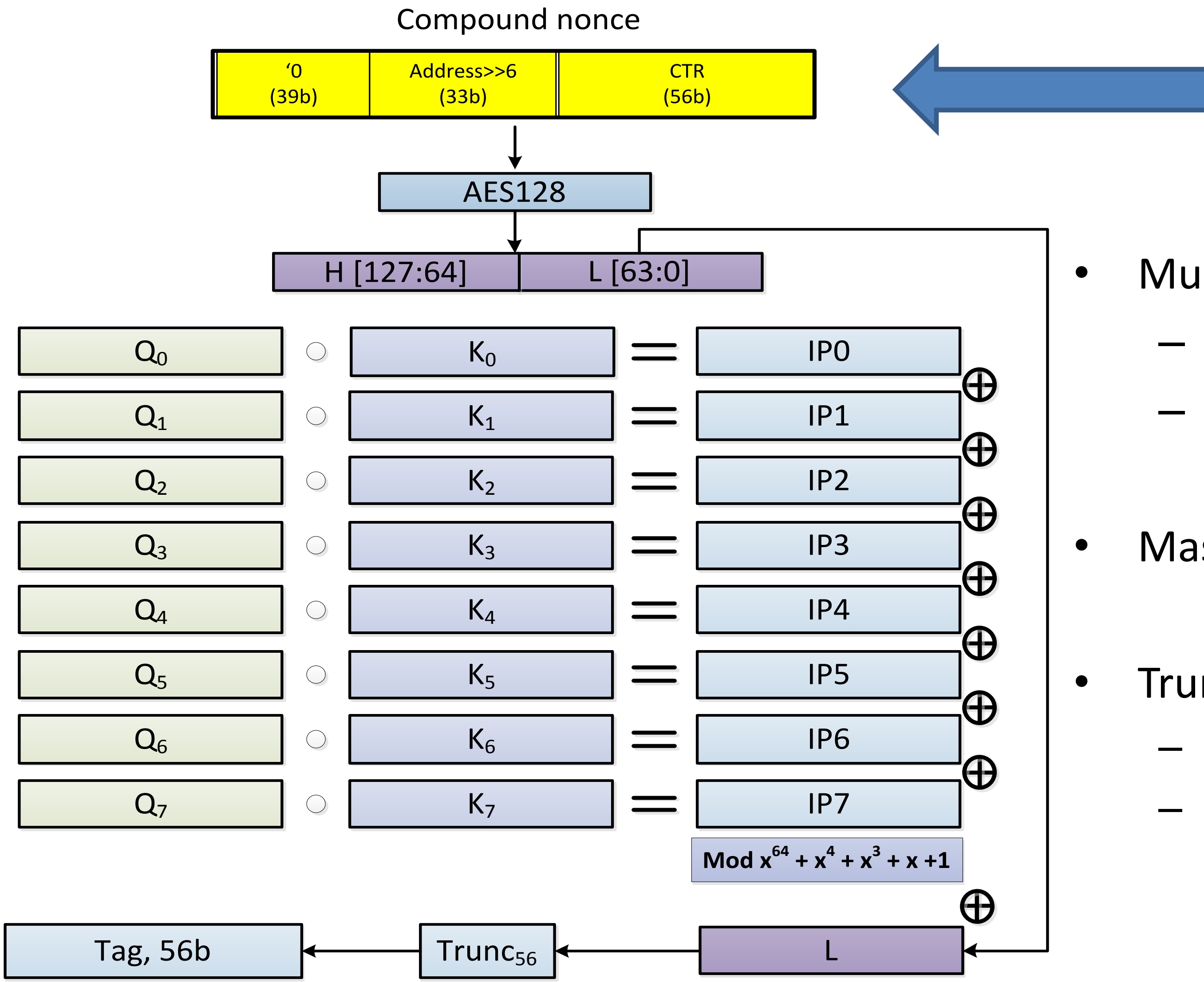
Address has 39 bits; idx: 2 bits representing location in the CL; Version: 56 bits



Encryption of 1 CL involves 4 AES operations

The MAC algorithm

Tag = L + $Q_0 \bullet K_0 + Q_1 \bullet K_1 + Q_2 \bullet K_2 + \ldots + Q_7 \bullet K_7$ in $GF(2^{64})$ Truncated to 56 bits



Spatial & temporal coordinates

- Multilinear universal hash
  - ("Inner Product hash")
  - Operations in GF (2^{64})
- Masked by (truncated) AES
- Truncated to 56 bits
  - Why? Real world…
  - If tags and counters have same length they can share same internal bus

The MEE cache

Sweetening the performance degradation impact

- Walking and processing the full read (write) flow for every cache miss can be very time-wise painful
  - E.g., 5 CL for "write": → [DATA, MAC, Version, L0, L2, L2 (L3)]

- Caching frequently used portions can significantly improve the performance
  - MEE internal cache holds counters and versions (not data nor data tags)
  - Counters that are retrieved from cache are trusted
    - Read/write flow stops at the cached node

  - With a lucky MEE-cache hit at the lowest level: Read operation required only one decryption and one MAC operation

What about security margins?

Aren’t 56-bit MAC tags

against the instinct of any cryptgrapher?

Maybe the 56-bit counters can be rolled over by dedicated attack code?

Worried?

Let’s define the super adversary model

The super  adversary model
idealized eavesdropper and forger

- Observes ciphertext / MAC tags samples (up to 2^{56})
  - Every observed ciphertext comes from a chosen plaintext
  - Every observed MAC tag comes from a chosen message
  - Spends 0 time (& cost) for storing all the data off platform
  - Collection all at 100% accuracy at highest (CL) granularity
  - **Collection time bounded only by platform's physical throughput**
- Then
  - Tries to gain information on plaintext (of victim applications)
  - Attempts a forgery (1 failure per key set) → reset and repeat

Beyond real world capabilities
but translates the discussion to a cryptographic problem

Some theorems on information theoretic bounds

Proposition 1 (Confidentiality bound). *Let* **Adv** *be the advantage of a probabilistic polynomial time algorithm in distinguishing the ciphertexts in* $\mathcal{T}'$ *from a set of random strings. Then,*

$$\mathbf{Adv} \leq \epsilon_{AES}(q') + \frac{(q')^2}{2^{125}} \tag{4}$$

Proposition 2 (The MEE forgery resistance). *An active adversary who collects a trace of* $q \leq 2^{56} - 2$ *message-tag samples that the MEE produces, and attempts a forgery, has success probability at most*

$$P_{success}(q) = \epsilon_{AES}(q) + \varepsilon \cdot \left(1 + \frac{q^2}{2^{128}}\right) \;\leq\; \epsilon_{AES}(2^{56}) + \frac{1}{2^{56}} \cdot \left(1 + \frac{1}{2^{16}}\right) \tag{13}$$

Translated to a "real world crypto" statement

- **Collecting many samples (even 2^{56}) does not give a significant advantage in distinguishing MEE ciphertexts from random**

- **Collecting many MAC tags samples (even 2^{56}) does not improve the forgery success probability beyond $1/2^{56}$ by any meaningful amount**

- **At 2^{56} samples the game is over (drop-and-lock enforced)**

Putting the crypto bounds to the test

How many samples can the adversary see?

- **Idealized: collection rate = platform's physical throughput**
  - Can he see 2^{56} ciphertexts?
  - Can he rollover 2^{56} counter?
  - Can he make ~2^{56} MAC tag guesses (try-fail-reboot-try…)
- Real system's limitation
  - AES engine throughput: 16B per cycle
  - Field multiplier throughput: 1 GF (2^{64}) multiply per cycle
  - 1 Write (CL + Tag) involves **at least** (with MEE internal cache hit)
    - **(4 + 1) AES operations + (8+2) field multiplications**
  - @ 2GHz (if overclocked)
- **Idealized sampling rate ≤ 1/10 freq. = 0.2G samples / sec**

Does an MEE with 56-bit tags and 56-bit counters give a sufficient security promise?

- Let's also assume 1000 "forge-boot" attempts per sec.
  - Above the CPU reset flow latency, but a nice number…

- Rollover (serial) would take at best 10.5 years
- Forgery (parallelizable) would take at best ~2M years

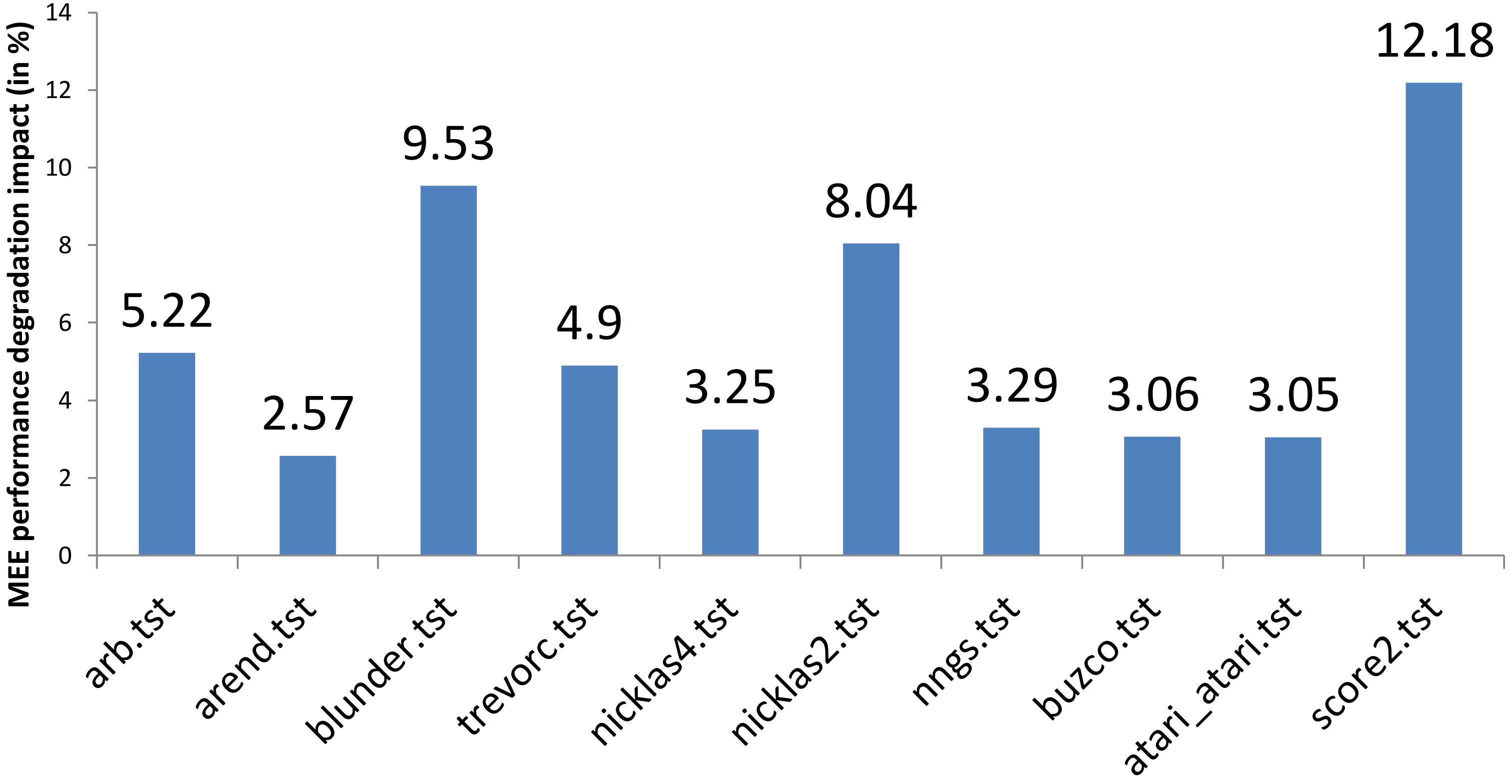
(or, 2 years over 1M machines doing forge-boot constantly)

Performance impact experiment

- Security costs ☹
- MEE overheads: encryption, authentication, tree walk…
- What is the observed performance impact on applications?
  - **The answer depends on multiple factors**
- Experiment:
  - 445.gobmk component of **SPECINT2006 v01**
  - Selecting 10 input files
  - Compiled the 445.gobmk test with Graphene (library OS), after adapting it to run inside an Intel SGX enclave.
  - This test measured (with the 10 input files) under two conditions:
    - **A.** without SGX (hence no MEE involved) **B.** inside an enclave (i.e., while MEE is active)
  - Comparison gives an estimation for the MEE performance impact

Performance estimation experimental results

MEE performance impact
between ~2.2% to ~12%, with average of ~5.5%



445.gobmk component of SPECINT 2006 (with 10 input files)
Bars show the performance degradation (in %) incurred by enabling the MEE

Conclusion

- MEE is essential to Intel® SGX technology
  - Provides data confidentiality, integrity, replay protection
- Building a real-word MEE in a real CPU is a formidable engineering challenge
  - MEE is based on a careful combination of tailored cryptographic primitives operating on a tailored integrity tree data structure
- Proven security margins even against an idealized adversary
- Reasonable (tolerable?) performance impact

- More information?
  - A detailed paper will be published
  - I am available for questions, comments and discussions

Thank you

Legal Disclaimer

- The comments and statements are mine and not necessarily Intel's
- Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.
- No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.
-