Memory Integrity Checking in AEGIS and Intel SGX

Marten van Dijk
Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

Department of Electrical & Computer Engineering
University of Connecticut

See next slide for sources from which we took material for slides.
Material Used:

Material taken from:
1. Intel SGX Tutorial (Reference Number: 332680-002) presented at ISCA 2015
4. S. Gueron, A Memory Encryption Engine Suitable for General Purpose Processors[J].
Outline

- Hash Tree for Memory Integrity Checking & Encryption
- AEGIS
- SGX Memory Encryption Engine (MEE)
- SGX Memory Access Protection
Checking external memory: Hash Tree (or Merkle Tree)

Secure

Insecure

Root hash
(k ~ 16-20 bytes)

m level 1 hashes

m² level 2 hashes

Data RAM (N ~ 1Gbyte)

By generalizing we can build a tree of hashes, which gives a better compromise.

- Secure Storage: k
- RAM overhead: 1/(m-1)
- Read/Write cost: log_m(N)

This slide has been taken from presentation by Blaise Gassend on “Caches and Hash Trees for Efficient Memory Integrity Verification”
Checking external memory: Hashing External Memory

Single hash can be used to protect the whole memory.

- Secure Storage: $k$
- RAM overhead: none
- Read/Write cost: $N$

This slide has been taken from a presentation by Blaise Gassend on “Caches and Hash Trees for Efficient Memory Integrity Verification”
Checking external memory: Multiple Hashes

p hashes (k ~ 16-20 bytes each)

Secure

Insecure

RAM (N ~ 1Gbyte) partitioned into p chunks

Read/write cost can be reduced by splitting the memory into chunks that have different hashes.

- Secure Storage: pk
- RAM overhead: none
- Read/Write cost: N/p

This slide has been taken from presentation by Blaise Gassend on “Caches and Hash Trees for Efficient Memory Integrity Verification”
Caching and Hash Trees: Naïve Approach

The simplest way of integrating hash trees into the memory hierarchy is to place all the hash tree machinery between two cache levels.

- Trusted side: checked data.
- Untrusted side: unchecked data and hashes.

Main problem:
- Each miss in L2 produces $\log_m(N)$ accesses to RAM.

This slide has been taken from presentation by Blaise Gassend on “Caches and Hash Trees for Efficient Memory Integrity Verification”
Caching Hashes: Integrated approach

- Much better performance if the hash tree machinery is integrated with L2 cache:
  - Hash tree machinery intercepts accesses to RAM.
  - Hash tree machinery reads and writes hashes it needs via L2.

- Big performance boost because:
  - Checked hashes are cached in trusted L2, so most accesses to external memory do not have to go to the root of the tree.

This slide has been taken from presentation by Blaise Gassend on "Caches and Hash Trees for Efficient Memory Integrity Verification"
Counter Mode Encryption

Hides decryption latency:
B[1] can be retrieved while the second AES decryption is in process, etc.
Outline

- Hash Tree for Memory Integrity Checking & Encryption
- AEGIS
- SGX Memory Encryption Engine (MEE)
- SGX Memory Access Protection
The Aegis secure processor relies on a security kernel in the operating system to isolate containers, and includes the kernel’s cryptographic hash in the measurement reported by the software attestation signature.

The Aegis memory controller encrypts the cache lines in one memory range, and HMACs the cache lines in one other memory range.

Aegis was the first secure processor not vulnerable to physical replay attacks, as it uses a Merkle tree construction to guarantee DRAM freshness.
SCM: Specialized HW (see MEE in the MC in SGX) that ensures protection of each process:
- Computes hash of program and data
- Assigns a Secure Process ID for on-chip memory access
- Performs memory integrity checking for off-chip memory access

AEGIS

Adversarial Model of AEGIS:
- Adversary can launch remote SW attacks and physical tampering of main memory
  - No attention is given to how observation of access patterns can leak sensitive information
  - No protection against the cache covert channel or other leakage from not properly flushing buffers
  - No protection against access to DRAM by peripherals
- HW TCB = CPU chip (with caches, mem. interface), Package
  - AEGIS forbids access by untrusted OS to reserved DRAM for Secure Containers (think Enclaves)
- SW TCB = App. Mod. (in Secure Enclave @ lowest priv. level), Sec. Kernel (@ highest priv. level in SMM) has no vulnerabilities
  - Allows multiple modules
  - Allows untrusted OS

Adversarial Models SGX and Sanctum

Adversarial Model SGX:
- Adversary can launch remote SW attacks and may actively alter/observe DRAM content (for the latter we need the MEE responsible for integrity checking and encryption):
  - The adversary is not attempting to derive information from access patterns to DRAM – either from actively observing the bus or leakage from page misses or from not properly flushing the branch history buffer (which can be easily fixed) or from the cache covert channel
- HW TCB = CPU chip (with caches, mem. interface), Package
  - Access to DRAM by peripherals is controlled by a trusted MC (as in Intel TXT) in the MEE on chip
  - SGX forbids access by untrusted OS to reserved DRAM for Secure Enclaves
- SW TCB = App. Mod. (in Secure Enclave @ lowest priv. level), SGX micro code (@ highest priv. level, higher than SMM’s level which includes BIOS)
  - Allows multiple modules
  - Allows untrusted OS

Adversarial Model of Sanctum:
- Adversary can only launch remote SW attacks
  - Because of HW isolation no encryption is necessary
- HW TCB = CPU chip (with caches, mem. interface), Package
  - Access to DRAM by peripherals is controlled by a trusted MCU (as in Intel SGX)
  - Sanctum forbids access by untrusted OS to reserved DRAM for Secure Enclaves
- SW TCB = App. Mod. (in Secure Enclave @ lowest priv. level), Sec. Monitor (@ highest priv. level)
  - Allows multiple modules (Sanctum prevents cache timing channel attacks using locality preserving cache-coloring)
  - Allows untrusted OS

Intel SGX follows AEGiS' blueprint!
Authentication

- The processor identifies security kernel by computing the kernel’s hash (on the `l.enter.aegis` instruction)
  - Similar to ideas in TCG TPM and Microsoft NGSCB
  - Security kernel identifies application programs

- `H(SKernel)` is used to produce a unique key for security kernel from a PUF response (`l.puf.secret` instruction)
  - Security kernel provides a unique key for each application

Message Authentication Code (MAC) ➔ A server can authenticate the processor, the security kernel, and the application
Protecting Program State

- On-chip registers and caches
  - Security kernel handles context switches and permission checks in MMU

Processor

External Memory

Memory Encryption: Counter-mode encryption

Integrity Verification: Hash trees
How should we apply the authentication and protection mechanisms?

What to protect?
- All instructions and data
- Both integrity and privacy

What to trust?
- The entire program code
- Any part of the code can read/write protected data
What Is Wrong?

- **Large Trusted Code Base**
  - Difficult to verify to be bug-free
  - How can we trust shared libraries?

- **Applications/functions have varying security requirements**
  - Do all code and data need privacy?
  - Do I/O functions need to be protected?
  - Unnecessary performance and power overheads

- **Architecture should provide flexibility** so that software can choose the **minimum** required trust and protection
Distributed Computation Example

DistComp()
{
    x = Receive();
    result = Func(x);
    key = get_puf_secret();
    mac = MAC(x,result,key);
    Send(result,mac);
}

Obtaining a secret key and computing a MAC
- Need both privacy and integrity

Computing the result
- Only need integrity

Receiving the input and sending the result (I/O)
- No need for protection
- No need to be trusted
AEGIS Memory Protection

Architecture provides five different memory regions
- Applications choose how to use

Static (read-only)
- Integrity verified
- Integrity verified & encrypted

Dynamic (read-write)
- Integrity verified
- Integrity verified & encrypted

Unprotected
- Only authenticate code in the verified regions

Memory Space

Receive(), Send() data
MAC() data
Func() data
Func(), MAC()
Receive(), Send()
Suspended Secure Processing (SSP)

Two security levels within a process
- Untrusted code such as Receive() and Send() should have less privilege

Architecture ensures that SSP mode cannot tamper with secure processing
- No permission for protected memory
- Only resume secure processing at a specific point
Certified Execution

**Job Dispatcher**

1. Compute $H(\text{Program})$
2. Program
3. Verify signature
4. Verify hash

**Processor**

3. Start: $\text{enter\_aegis}$: Processor computes $h = H(\text{Program})$
4. Execute $\rightarrow$ Out
5. Sign: $\text{sign\_msg}$
6. $\{h, \text{Out}\}_{SK_p}$, $\{PK_p\}_{SK_m}$

---

Digital Rights Management

Content Provider
- Secure Player, Content, PKm
  ① Compute $H(\text{Player})$
  ② Secure Player
  ④ Set up SSL connection
  ⑤ Order and deliver content

Customer’s Processor
- SKp, PKp, ${\{PKp\}}_{SKm}$
  ③ Start PTE: enter_aegis
    - set_aegis_mode
    - (uses sign_msg)
  ⑥ Decode and play
  ⑦ Analog Output

Implementation

Fully-functional system on an FPGA board

- AEGIS (Virtex2 FPGA), Memory (256MB SDRAM), I/O (RS-232)
- Based on openRISC 1200 (a simple 4-stage pipelined RISC)
- AEGIS instructions are implemented as special traps
Area Estimate

- **I/O (UART, SDRAM ctrl, debug unit)**: 0.258mm²
- **Cache (16KB)**: 1.050mm²
- **IV Unit (5 SHA-1)**: 1.075mm²
- **Encryption Unit (3 AES)**: 0.864mm²
- **Cache (4KB)**: 0.504mm²
- **Code ROM (11KB)**: 0.138mm²
- **Scratch Pad (2KB)**: 0.261mm²
- **PUF**: 0.027mm²
- **Core**: 0.086mm²
- **I-Cache (32KB)**: 1.815mm²
- **D-Cache (32KB)**: 2.512mm²

**Synopsys DC with TSMC 0.18u lib**

New **instructions** and **PUF** add 30K gates, 2KB mem (1.12x larger)

**Off-chip protection** adds 200K gates, 20KB memory (1.9x larger total)

The area can be further optimized
Performance Slowdown

Performance overhead comes from off-chip protections

Synthetic benchmark
- Reads 4MB array with a varying stride
- Measures the slowdown for a varying cache miss-rate

Slowdown is reasonable for realistic miss-rates
- Less than 20% for integrity
- 5-10% additional for encryption

<table>
<thead>
<tr>
<th>D-Cache miss-rate</th>
<th>Slowdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integrity</td>
</tr>
<tr>
<td>6.25%</td>
<td>3.8</td>
</tr>
<tr>
<td>12.5%</td>
<td>18.9</td>
</tr>
<tr>
<td>25%</td>
<td>31.5</td>
</tr>
<tr>
<td>50%</td>
<td>62.1</td>
</tr>
<tr>
<td>100%</td>
<td>130.0</td>
</tr>
</tbody>
</table>
5 EEMBC kernels and 1 SPEC benchmark

EEMBC kernels have negligible slowdown
- Low cache miss-rate
- Only ran 1 iteration

SPEC twolf also has reasonable slowdown

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown (%)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integrity</td>
<td>Integrity + Privacy</td>
<td></td>
</tr>
<tr>
<td>routelookup</td>
<td>0.0</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>ospf</td>
<td>0.2</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>autocor</td>
<td>0.1</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>conven</td>
<td>0.1</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>fbital</td>
<td>0.0</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>twolf (SPEC)</td>
<td>7.1</td>
<td>15.5</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Hash Tree for Memory Integrity Checking & Encryption
- AEGIS
- SGX Memory Encryption Engine (MEE)
- SGX Memory Access Protection
Memory Encryption Engine

- Memory Encryption Engine (MEE):
  - Added in the uncore part of the processor (Memory Controller)
  - Protects SGX’s Enclave Page Cache against the following physical attacks:
    - Data Confidentiality: Collections of memory images of DATA written to the DRAM cannot be distinguished from random data.
    - Integrity + freshness: DATA read back from DRAM to LLC is the same DATA that was most recently written from LLC to DRAM.
How the MEE works – in a nutshell

- Core issues a transaction
  - (to MEE region); e.g., WRITE
- Transaction misses caches and forwarded to Memory Controller
- 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

Unauthorized DMA transactions that target the PRM (Processor Reversed Memory) range should be aborted.
MEE basic setup and policy

- Memory access always at 512 bits Cache Line (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)

Encryption Key: 128 bits
MAC 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

COUNTER_BLOCK

- '0' (37b)
- PhysAdr[38:6] (33b)
- idx (2b)
- Version Ctr (56b)

CONFIDENTIALITY_KEY

AES128

Encrypted Counter Block

Plaintext (or ciphertext), 128b

XOR

Ciphertext (or plaintext), 128b

Encryption of 1 CL involves 4 AES operations
Message Authentication Code

- MAC can be used to protect memory integrity.
- But what is the problem if we only use MAC?

- Replay attack

- Solutions:
  1. Hash Tree (Store updated root hash in TCB)
     - One root hash for the whole memory
  2. Stateful MAC (Store updated states in TCB)
     - One state for each cache line
     - How to store all the states efficiently???
One level data structure

Tag = MAC (CTR, CL)

CTR is trusted

Integrity + freshness

Too many counters in trusted region. Too 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 into counter cache line to save the memory accesses.

Why don’t we embed tags into data cache lines as well?
A Counter Cache Line

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

\[ 56 \times 8 + 56 + 8 = 512 \]
What is the compression rate?
The overall compression rate

SRAM
Counters Level 3: 3KB

Embedded tags

Versions & tags

Embedded MEE-MAC

Counters Level 2: 24KB

Counters Level 1: 192KB

1.5MB

MAC tags & Counters: 12 + 12 = 24MB

DATA: 96MB

Embedded tags

Embedded tags
Comparison with Hash Tree

Access this data
The MAC algorithm

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

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
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)
Outline

- Hash Tree for Memory Integrity Checking & Encryption
- AEGIS
- SGX Memory Encryption Engine (MEE)
- SGX Memory Access Protection
SGX Memory Access Protection

- MEE sits in MC, it cannot protect an enclave’s memory from software attacks.

- The root of SGX’s protections against software attacks is memory access checks which prevents the currently running software from accessing memory that does not belong to it.

- Implemented in Page Miss Handler (PMH)
  - PMH triggers the extra microcode for all address translations
  - All the SGX instructions are implemented in microcode, which introduces many new registers for storing metadata of enclave.
Security Check for Memory Access

SGX adds a few security checks to the PMH. The checks ensure that all the TLB entries created by the address translation unit meet SGX's memory access restrictions.
SGX Security Check Correctness

- Top-level invariant: At all times, all the TLB entries in every logical processor will be consistent with SGX’s security guarantees.

- First breakdown the top level invariant into three cases on:
  - whether a logical processor (LP) is executing enclave code or not
  - whether the TLB entries translate virtual addresses in the current enclave's ELRANGE
Case Invariants

1. At all times when an LP is outside enclave mode, its TLB may only contain physical addresses belonging to DRAM pages outside the PRM.

2. At all times when an LP is inside enclave mode, the TLB entries for virtual addresses outside the current enclave’s ELRANGE must contain physical addresses belonging to DRAM pages outside the PRM.

3. At all times when an LP is in enclave mode, the TLB entries for virtual addresses inside the current enclave’s ELRANGE (Enclave Linear Address Range) must match the virtual memory layout specified by the enclave author.
Invariant 1

- At all times when an LP is outside enclave mode, its TLB may only contain physical addresses belonging to DRAM pages outside the PRM.
Invariant 2

At all times when an LP is inside enclave mode, the TLB entries for virtual addresses outside the current enclave’s ELRANGE (Enclave Linear Address Range) must contain physical addresses belonging to DRAM pages outside the PRM.
Invariant 3

At all times when an LP is in enclave mode, the TLB entries for virtual addresses inside the current enclave’s ELRANGE (Enclave Linear Address Range) must match the virtual memory layout specified by the enclave author.
The entire flow

Top-level invariant: At all times, all the TLB entries in every logical processor will be consistent with SGX's security guarantees.

Invariant 1

Invariant 2

Invariant 3