SeM: A CPU Architecture Extension for Secure Remote Computing

Ofir Shwartz, Yitzhak Birk

Hardware and Architectural Support for Security and Privacy (HASP) 2017
Motivation

- Clouds are promising
  - Pay per use
  - No overhead costs
  - Establish and discard resources on the fly

- Security limits adoption
  - Risks at many levels
  - Software: other users (competitors), OS, hypervisor, VMM
  - Privileged attacker: exploit bugs, cloud owner
  - Hardware: physical attacks
Threat Model

- **Platform software**
  - Hypervisor, VMM, OS are **untrusted**
  - Any management software is **untrusted**

- **Platform hardware**
  - Memory, network, board signals are **untrusted**
  - CPU is **trusted** – not internally snooped or modified

- **An attacker has full control of the machine**
  - Can implant software or hardware before or during the operation of the program
Previous Works

- **Software based:**
  - Easy to adopt, no hardware changes are required
  - Some software must be trusted; untrusted cloud owner?
    - Exceptions: software based, hardware verified
  - Software performs security tasks – performance overheads

- **Hardware:**
  - Commonly: only the CPU is trusted.
  - Many do not support existing binaries, and performance is low
  - Intel SGX
    - Only matches programs developed for it
    - Limited performance
  - Software on top of SGX:
    - E.g., Haven, PANOPLY, Graphene, SCONE, …
    - Support for some applications, still performance issues
Goals

• Keep confidentiality and integrity of
  ▫ Data: input, temporary, output
  ▫ Code
  ▫ State of execution

• While also:
  ▫ Support existing applications (binaries)
  ▫ Support conventional systems: multi-tasking, interrupts, signals, system calls, etc.
  ▫ High performance execution
  ▫ Low power / area overheads
Explicitly, How to..

• Problem #1: Protect code and data
• Problem #2: Protect state and flow
• Problem #3: Using untrusted code
• Problem #4: Thread management
• Problem #5: Multi-node integrity
Secure Machine (SeM) Arch. Ext.
Problem #1: Protect Code and Data

- **Common approach: memory encryption**
  - Code and data: signed and encrypted when in untrusted memory, clear when in cache
  - Counter mode encryption (e.g., GCM)
  - Signatures (e.g., GHASH) and a hash tree

- **Key Storage: securely store secret keys**
  - Per process, or group of processes
  - Keys: write-only for software to form a Key Entry
    - By using public key cryptography
  - Upon start, attach with the process ID(s) – details in the paper

- **But what about cached data?**
  - Main idea: couple instructions and data by a security domain
Memory ⇔ Cache

- On cache miss: fetch block from memory
  - Decrypt and validate
  - If validated correctly, fetch decrypted block
  - else, fetch original
- To cache: data, Auth (true/false), owner ID (ID in the Key Storage)

- Cache blocks:
  - Each block also has Auth bit and OID
  - \{Auth, OID\} serves as the **Security Domain** of the block

<table>
<thead>
<tr>
<th>Cache</th>
<th>Auth</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>d/m.d.</td>
<td>true</td>
<td>23</td>
</tr>
<tr>
<td>d/m.d.</td>
<td>false</td>
<td>60</td>
</tr>
</tbody>
</table>
Cache $\Rightarrow$ Memory

- **Upon eviction:** If $\text{Auth}=t$, sign and encrypt
  - Using the keys in the Key Storage (for owner ID)
  - Also update the integrity structure
- **Else:** evict as is
Cache ↔ Exec Unit

- On instruction fetch, also fetch Auth and OID

- Secure Access
  - On each cache access (memory instruction - load, store,..)
  - If \( \text{inst} \{ \text{Auth}, \text{OID} \} = \text{data} \{ \text{Auth}, \text{OID} \} \)
  - allow access
  - else
  - reject
## Secure Access

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Data</th>
<th>Auth</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata</td>
<td>Load R1, [20]</td>
<td>True</td>
<td>30</td>
</tr>
<tr>
<td>metadata</td>
<td>0xABCD</td>
<td>True</td>
<td>30</td>
</tr>
<tr>
<td>metadata</td>
<td>0x5678</td>
<td>false</td>
<td>30</td>
</tr>
<tr>
<td>metadata</td>
<td>0x1122</td>
<td>true</td>
<td>35</td>
</tr>
</tbody>
</table>

**Cache Miss!**

**SMU Validate**

**Untrusted Memory**

**Grant!**

**Load R1, [60]**

**Instruction Fetch**

**Execution Unit**

**0xABCD**
Secure Access: Benefits

- **Safe**: foreign code cannot validate correctly
  - Even if privileged
  - Must be validated to access validated (protected) data
- **Automatic** boundary between trusted and untrusted worlds
  - Unmodified code *cannot* expose memory data or import unauthorized memory data by mistake
- **Performance**: adversarial blocks co-reside in the cache
  - No added evictions on top of a regular machine
  - The system matches the performance the plain memory encryption subsystem in use (encrypt and sign) (~2%)
Special Memory Instructions

- Must be validated to run:
  - *StoreNA* – store and set Auth=false
    - Send data to untrusted code
  - *LoadNA* – load from a block with Auth=false
    - Read data from untrusted code
  - *InitA* – store zeros to a memory region, sign correctly
    - Initialize newly allocated memory
Problem #2: Protect State and Flow

- State: register values; Flow: seq. of instructions

- Example: Interrupt issues an untrusted instruction unexpectedly
  - Register values are exposed (*secret context*)
  - When back, need to enforce correct register values and correct instruction
Security Modes (Cache ↔ Exec Unit)

- Work in two modes: trusted and untrusted
  - Trusted mode: only runs validated ($Auth=t$) instructions
  - Untrusted mode: only runs non-validated instructions
  - Switch automatically
- If Trusted and $\text{inst}\{\text{Auth}\}=$false
  - Store reg values in SMU Sealed Storage (SSS) and clear (secret context), keep the next legal entry point (LEP)
  - Change to Untrusted mode
- If Untrusted and $\text{inst}\{\text{Auth}\}=$true, and the process has a secret context in the SSS (and $\text{inst}\{\text{address}\}$==LEP)
  - Restore the secret context
  - Change to Trusted mode
SMU Sealed Storage

• May store secret context of one or more programs
  ▫ Can be implemented using a register window (~1 clock cycle for switch)

• Upon context switch, may store content into the program’s memory space
  ▫ Takes ~40 cycles on top of ~2k cycles of C.S.
  ▫ Protected automatically by memory encryption
  ▫ Triggered by a watchdog for changing the page table register (microcode)
Stack Management

- Untrusted code (on behalf of the secure process) require an accessible stack
- Use two stacks: Secret and Non-secret
  - The secret stack is signed and encrypted - used for the trusted code (by conventional memory instructions)
  - The non-secret is clear - used by untrusted code
- Stack pointer is switched with the secure state switch
  - Secure stack is automatically created and initialized by the program –details in the paper
Problem #3: Using Untrusted Code

- Library functions: embed into the binary, when preparing for SeM
  - Becomes trusted
- System calls are still required

- Solution: use new syscallX instructions that keep a set of registers untouched on switch to untrusted
  - Replace original syscall instructions on preparing for SeM
  - Static analysis to determine the system call needs – details in the paper
SeM-Prepare

- Input: a compiled binary
- Instrumenting the binary for preparing it for the cloud (deployment)
  - Statically embed shared libraries
  - Attach itself with the Key Storage entry
  - Allocate and initialize (InitA) the secure stack
  - Initialize memory on allocation
  - Replace syscall instructions with syscallX
  - IO accesses: enc and dec by software (wrap syscalls)
- When done, encrypt and sign
Evaluation

- SPEC CPU 2006 benchmark suite
- Prepared by SeM-Prepare
- Evaluated by SeM-Simulator
  - Memory encryption
  - Secure Access enforcement
  - Security modes and register switch
  - Support new SeM instructions: memory, system calls.
- Purpose: prove **applicability** and measure **performance**
Results

- %Reduction with Mem Enc
- %Reduction without Mem Enc

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>%Reduction with Mem Enc</th>
<th>%Reduction without Mem Enc</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h264ref</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hmmer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libquantum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>perlbench</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sjeng</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalancbmk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>omnetpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results

<table>
<thead>
<tr>
<th></th>
<th>astar</th>
<th>bzip2</th>
<th>gcc</th>
<th>h264ref</th>
<th>hmer</th>
<th>libquantum</th>
<th>mcf</th>
<th>perlbench</th>
<th>sjeng</th>
<th>xalancbk</th>
<th>omnetpp</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Reduction with Mem Enc</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
</tr>
<tr>
<td>%Reduction without Mem Enc</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
<td>1.9%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
Results

- %Reduction with Mem Enc
- %Reduction without Mem Enc
- Total IO (100B for 1M instructions)
- Total Mem Allocation (10B for 1k instructions)
Conclusions

- **SeM is a secure architecture extension**
  - Can be easily added to existing CPU architectures
  - Supports existing binaries – automatically instrumented
  - Negligible area (~0.01%) and performance (~2%) costs
- **SeM monitors memory ↔ cache and cache ↔ execution unit**
  - Hardware separation between different security domains
  - Based on simple in-cache metadata \{Auth, OID\}
  - Protect the context and flow of the secure application
- **Ongoing work (advanced stages)**
  - Secure multi-threading and multi-node computation (incl. heterogeneous multi-node integrity)
Additional Slides
Security Management Unit (SMU)

- Encryption path
  - To untrusted Memory
  - From cache (WB)
- Key table
- Key generator
- RSA
- SMU Instructions FE
  - Instructions and return values
  - SMU Sealed Storage
    - To cache (Fetch)
    - CPU regs
      - Cache