A Secure and Formally Verified Commodity Multiprocessor Hypervisor

Shih-Wei Li (黎士瑋)
Assistant Professor
CSIE Department @ National Taiwan University
2021/10/01
Todays’ Talk
A Secure and Formally Verified Commodity Multiprocessor Hypervisor

• “Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor”, Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, John Zhuang Hui, Proceedings of the 30th USENIX Security Symposium (Usenix Security 2021)


Flocking to the Cloud

By 2025, 85% of enterprises will have a cloud-first principle.

Sean Gallup/Getty Images
Virtualization
Hypervisor Complexity = Hypervisor Bugs
Hypervisor Bugs

A malicious host or VM user that exploits hypervisor bugs results in:

- Code execution:
  - CVE-2020-16891, CVE-2019-0721
- Privilege escalation:
  - CVE-2020-1047, CVE-2018-18021
- Information leakage:
Hypervisor Bugs = Security Risks

- Attackers that exploit the hypervisor can gain full access to VM data
Formal Verification

- Verify the hypervisor contains no vulnerabilities and protects VM data
  - Includes three components: implementation, specification, hardware model
- Soundness of the proof relies on the accuracy of the hardware model
Formal Verification

• Previous work verified simplistic systems with limited functionality
  
  • seL4 and CertiKOS do not support common hypervisor features and were verified using simplistic hardware models
Formal Verification

- seL4 and CertiKOS requires significant effort to verify few thousand LOC

Few K LOC

2M+ LOC
Formal Verification

- seL4 and CertiKOS requires significant effort to verify few thousand LOC

Few K LOC

2M+ LOC

Existing verification approaches are intractable for commodity hypervisors
Hypothesis:

Modest changes to commodity systems can make it possible to verify their security properties
Microverification

- Retrofit a commodity system to enable formal verification of security properties
  - Decompose the system into a small core and a set of untrusted services
  - Prove the properties of the entire system by verifying the core alone
Microverification of KVM

- Prove KVM protects VM *confidentiality* and *integrity* while preserving KVM’s overall feature set and performance
Agenda

• A Secure and Formally Verified Commodity Multiprocessor Hypervisor
  • Retrofitting KVM
    • Verifying KVM
  • Summary
• Future Work
Retrofitting KVM
Overall Design

• Retrofit the monolithic KVM hypervisor into SeKVM with two parts:
  • Small **KCore** that protects VM data
  • Large set of *untrusted* services (**KServ**) to provide complex functionalities
Retrofitting KVM

Observations

• Applications increasingly adopt end-to-end approach to protect their I/O data
• Many hypervisor functions do not need access to VM data
  • Ex: VCPU scheduling and memory allocation
Retrofitting KVM

General Principles

• **KCore** has full hardware access
  - Protects VM data in CPU registers and memory
• **KServ** has restricted hardware access
  - Provides functionality that requires no access to unencrypted VM data
• Relies on VMs protect their I/O data
  - Employ encrypted networking (TLS/SSL) or virtual disk
Retrofitting KVM
Functionality Split

• **KCore** provides
  • CPU virtualization, page table management
  • Access control for VM data in CPU and memory

• **KServ** provides
  • I/O virtualization
  • Resource allocation and scheduling
Retrofitting KVM
Leveraging Hardware Virtualization Extensions

• SeKVM leverages Arm virtualization extensions to:
  • Deprivilege KServ in EL1
  • Run KCore in EL2 to protect VM data
  • Control hardware features for VM protection
Memory Protection

- KCore leverages Nested Page Tables (NPTs)
- Use the nested level translation to enforce memory access control
Memory Protection

• KCore manages NPT for KServ to control KServ’s memory accesses
  • Ensure KServ’s NPT does not map to KCore’s or VM’s private memory
Memory Protection

- SeKVM tasks KServ to provide memory allocation
  - Use identity mapping in KServ’s NPT to reuse KServ’s functionality
CPU Protection

- KCore ensures KServ cannot access VM CPU data on the hardware
  - Interpose switches between the VMs and KServ
  - Context switch VM’s CPU registers to KCore’s private memory
Example: Handling VM NPT Page Fault

- KCore controls the hardware to trap NPT page faults to itself
- KCore saves VM CPU from the hardware and switches to KServ for memory allocation
Example: Handling VM NPT Page Fault

- KCore unmaps newly allocated memory from kNPT and maps it to VM’s NPT
- KCore restores VM CPU to the hardware before entering VM
Agenda

• A Secure and Formally Verified Commodity Multiprocessor Hypervisor
  • Retrofitting KVM
  • Verifying KVM

• Summary

• Future Work
Verifying KVM
Verify Functional Correctness of KCore

- Ensure that the concurrent KCore’s implementation contains no vulnerabilities
- Show KCore’s implementation refines its specification

![Diagram of KCore and KServ layers]
Verifying KVM
Verify Security Properties of SeKVM

• Verify security properties hold for the entire KVM hypervisor

• Reason over KCore’s specification and show that VM data is protected for any implementation of KServ and VMs interacting with KCore
Verify Functional Correctness of KCore
Layered Verification Approach

• Modularize KCore’s implementation and proofs into layers
  • Decompose KCore into simpler components to reduce verification effort
Verify Functional Correctness of KCore
Layered Verification Approach

• Allow higher layer functions to only use lower layer functions
  • Verify higher layers using the specification of the verified lower layers

Diagram:
- KServ
- SeKVM
- Linux VM
- KCore
- Hardware
- L0 Specification
  - L1 Specification
    - L1 Implementation
    - L0 Specification
      - L0 Implementation
      - Refinement
      - Refinement
Verify Functional Correctness of KCore
Layered Hardware Model

• Introduce a layered hardware model that accounts for multiprocessor hardware features

• Tailor the complexity of the hardware model for the software needs
Verify Functional Correctness of KCore
Layered Hardware Model
Verify Functional Correctness of KCore
Layered Hardware Model

• Verify KCore using a layered hardware model that accounts for multi-level shared page tables, tagged TLBs, and writeback caches

• Structure KCore’s layered modules to match the appropriate layered model
Verify Functional Correctness of KCore Refinement

• Verify that KCore’s implementation refines its specification
  • Incrementally prove that KCore’s top-layer is refined by a stack of layers
Refinement in Multiprocessor Settings
Case Study: NPT Updates

- A multiprocessor VM has a shared NPT

```c
void set_npt(u32 vmid, u32 gpa, u32 pa) {
    acq_lock_npt(vmid);
    u32 pte = pt_load(vmid, pgd_offset(gpa));
    pt_store(vmid, pte_offset(pte, gpa), pa);
    rel_lock_npt(vmid);
}
```
Refinement in Multiprocessor Settings
Case Study: NPT Updates

- Multiprocessor VMs can read the shared NPT via the MMU anytime
  - Concurrent NPT reads within the critical section

```c
void set_npt(u32 vmid, u32 gpa, u32 pa) {
    acq_lock_npt(vmid);
    u32 pte = pt_load(vmid, pgd_offset(gpa));
    pt_store(vmid, pte_offset(pte, gpa), pa);
    rel_lock_npt(vmid);
}
```
Refinement in Multiprocessor Settings
Case Study: NPT Updates

- A buggy set_npt implementation could result in information leakage

```c
void set_npt_insecure(u32 vmid, u32 gpa, u32 pa) {
    acq_lock_npt(vmid);
    u32 pte = pt_load(vmid, pgd_offset(gpa));
    //pa1 maps to a page by the VM vmid1
    pt_store(vmid, pte_offset(pte, gpa), pa1);
    pt_store(vmid, pte_offset(pte, gpa), pa);
    rel_lock_npt(vmid);
}
```
Refinement in Multiprocessor Settings
Case Study: NPT Updates

• Show the set_npt implementation refines an atomic specification

```c
void set_npt(u32 vmid, u32 gpa, u32 pa) {
    acq_lock_npt(vmid);
    u32 pte = pt_load(vmid, pgd_offset(gpa));
    pt_store(vmid, pte_offset(pte, gpa), pa);
    rel_lock_npt(vmid);
}
```

**set_npt spec**: map gpa to pa in VM vmid’s NPT
Refinement in Multiprocessor Settings
Case Study: NPT Updates

- Previous approaches do not distinguish between the correct and incorrect implementation
- Refine both of the implementations to the same atomic specification
- Hide information leakage in the incorrect implementation

**set_npt specification**: map gpa to pa in VM vmid’s NPT

```
set_npt specification
  ↑  ↑
Refinement  Refinement
set_npt Implementation
  ↑
set_npt_insecure Implementation
```
Security-preserving Layers

• Ensure refinement doesn’t hide information leakage and preserves security properties
Security-preserving Layers

• Employ *transparent trace refinement* to ensure the implementation reveals at most as much information as its specification

• Only refines the correct implementation to the atomic specification

• Verify KCore correctly manages shared page tables

*set_npt spec*: map gpa to pa in VM vmid’s NPT
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

• TLB caches page table translations
• Arm provides tagged TLB to avoid flushes when switching CPU execution
  • Software flushes TLB when updating page tables
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Consider the TLB caches entries from NPT — translate a guest physical page (gfn) to a physical page (pfn)

1. flush_tlb(pfn:1, A)
2. unmap(pfn:1, A)
3. map(pfn:1, B)
Verify SeKVM using layered hardware model
Case Study: Verify KCore’s TLB Management

- Multiprocessor VM A that accesses pfn 1 results in caching of pfn 1’s mapping in the TLB

VM A accesses pfn 1

TLB miss, refill from NPT
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

• VM A can access pfn 1 through the TLB and breaks VM isolation

![Diagram showing TLB and VM interactions](image)

- **t0**: unmap(pfn:1, A)
- **t1**: flush_tlb(pfn:1 A)
- **t2**: unmap(pfn:1, A)
- **t3**: map(pfn:1, B)
- **t4**: access pfn 1
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Verify KCore’s code that manages TLBs using a hardware model with tagged TLB behaviors

- Refine the complex model with TLBs and page tables into the simpler model with only page tables

- Verify KCore’s code that does not manage TLBs using a simpler model
Verify SeKVM using layered hardware model
Case Study: Verify KCore’s TLB Management

- Intuition: Pages observable through the incorrectly managed TLB will a superset of the ones through page tables
- The TLB may include stale entries if not flushed after page table updates

VM A accesses pfn 1 before unmap

1. flush_tlb(pfn:1, A)
2. unmap(pfn:1, A)
3. map(pfn:1, B)
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Introduce **page observers** — the set of principals (VMs or KServ) who can observe a physical page (pfn) through TLBs or page tables

- Merge consecutive page observers into **page observer groups**
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Consider the following execution steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Page observers TLB</th>
<th>Page observers PT</th>
<th>Page observer groups TLB</th>
<th>Page observer groups PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unmap(pfn:1, A)</td>
<td>{1: A}</td>
<td>{1: A}</td>
<td>{1: A}</td>
<td>{1: A}</td>
</tr>
<tr>
<td>2</td>
<td>flush_tlb(pfn:1, A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>map(pfn:1, B)</td>
<td>{1: A}</td>
<td>{1: __}</td>
<td>{1: A}</td>
<td>{1: A}, {1: __}</td>
</tr>
</tbody>
</table>
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Consider the following execution steps

1. `unmap(pfn:1, A)`
2. `flush_tlb(pfn:1, A)`
3. `map(pfn:1, B)`

<table>
<thead>
<tr>
<th>Page observers</th>
<th>Page observers</th>
<th>Page observer groups</th>
<th>Page observer groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>PT</td>
<td>groups TLB</td>
<td>groups PT</td>
</tr>
<tr>
<td>{1: A}</td>
<td>{1: A}</td>
<td>{1: A}</td>
<td>{1: A}</td>
</tr>
<tr>
<td>{1: A}</td>
<td></td>
<td>{1: __}</td>
<td>{1: A}</td>
</tr>
<tr>
<td>{1: __}</td>
<td></td>
<td>{1: __}</td>
<td>{1: A}, {1: __}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{1: A}, {1: __}</td>
</tr>
</tbody>
</table>
Verify SeKVM using layered hardware model
Case Study: Verify KCore’s TLB Management

- Consider the following execution steps

1. `unmap(pfn:1, A)`
2. `flush_tlb(pfn:1, A)`
3. `map(pfn:1, B)`

<table>
<thead>
<tr>
<th>Page observers</th>
<th>TLB</th>
<th>Page observers</th>
<th>PT</th>
<th>Page observer groups TLB</th>
<th>Page observer groups PT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{1: A}</td>
<td></td>
<td>{1: A}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{1: A}</td>
<td></td>
<td>{1: __}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{1: __}</td>
<td></td>
<td>{1: __}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{1: B}</td>
<td></td>
<td>{1: B}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Prove KCore correctly manages the TLBs by showing that TLBs and page tables produce the same sequence of page observer groups

1. `unmap(pfn:1, A)`
2. `flush_tlb(pfn:1, A)`
3. `map(pfn:1, B)`

<table>
<thead>
<tr>
<th>Page observers</th>
<th>Page observers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLB</strong></td>
<td><strong>PT</strong></td>
</tr>
<tr>
<td>{1: A}</td>
<td>{1: A}</td>
</tr>
<tr>
<td>{1: A}</td>
<td>{1: __}</td>
</tr>
<tr>
<td>{1: __}</td>
<td>{1: __}</td>
</tr>
<tr>
<td>{1: B}</td>
<td>{1: B}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page observer groups</th>
<th>Page observer groups</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLB</strong></td>
<td><strong>PT</strong></td>
</tr>
<tr>
<td>{1: A}</td>
<td>{1: A}</td>
</tr>
<tr>
<td>{1: A}</td>
<td>{1: __}</td>
</tr>
<tr>
<td>{1: A}, {1: __}</td>
<td>{1: A}, {1: __}</td>
</tr>
<tr>
<td>{1: A}, {1: __}, {1: B}</td>
<td>{1: A}, {1: __}, {1: B}</td>
</tr>
</tbody>
</table>

Same
Verify SeKVM using layered hardware model

Case Study: Verify KCore’s TLB Management

- Use this approach to detect incorrect TLB management

1. `flush_tlb(pfn:1, A)`
2. `unmap(pfn:1, A)`
3. `map(pfn:1, B)`

<table>
<thead>
<tr>
<th>Page observers</th>
<th>Page observers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>PT</td>
</tr>
<tr>
<td><img src="chart.png" alt="Diagram" /></td>
<td><img src="chart.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Can be refilled after TLB flush

Different
Verify SeKVM using layered hardware model

- Verify KCore’s lower layered software using the detailed hardware model refines higher layered software with the simpler abstract hardware model.
- Verify higher layered software using the abstract hardware model.
Verify Security Properties of SeKVM

• Verify SeKVM’s protection of VM data over KCore’s top-layer specification for all principals, *KServ*’s or *VMs*’ interaction with KCore
Verify Security Properties of SeKVM

Formulating Security Properties as Noninterference

- Formulate the confidentiality and integrity guarantees in terms of noninterference.
- Intuition: one principal’s execution will not infer or affect others’ data.
Verify Security Properties of SeKVM
Formulating Security Properties as Noninterference

• Formulate the confidentiality and integrity guarantees in terms of **noninterference**

• Intuition: one principal’s execution will not infer or affect others’ data

---

Proving noninterference for VM A
Verify Security Properties of SeKVM
Formulating Security Properties as Noninterference

- Formulate the confidentiality and integrity guarantees in terms of noninterference
- Intuition: one principal’s execution will not infer or affect others’ data

Proving noninterference for VM A
Verify Security Properties of SeKVM
Formulating Security Properties as Noninterference

• Formulate the confidentiality and integrity guarantees in terms of *noninterference*

• Intuition: one principal’s execution will not infer or affect others’ data

noninterference for VM A ensures VM B’s data is confidential to VM A
Verify Security Properties of SeKVM
Formulating Security Properties as Noninterference

- Formulate the confidentiality and integrity guarantees in terms of noninterference
- Intuition: one principal’s execution will not infer or affect others’ data

noninterference for VM A ensures VM A’s data cannot be modified by VM B, retaining VM A’s integrity
Verify Security Properties of SeKVM
Proving Noninterference over observation

• A principal’s observation: subset of the machine state that contains a principal’s data — include a principal’s CPU registers and memory

• Prove noninterference for each principal using the observation
Implementation

Verifying VM Confidentiality and Integrity of SeKVM

- Verify SeKVM protects the confidentiality and integrity of VM data
- Modularize KCore implementation into 34 security preserving layers

<table>
<thead>
<tr>
<th>KCore</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>verified C and ASM code</td>
<td>3.8K</td>
</tr>
<tr>
<td>HACL*</td>
<td>10.1K</td>
</tr>
</tbody>
</table>
Implementation

Proof Effort

• Proofs for SeKVM are written in Coq and machine-checked

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of Coq code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top layer specifications</td>
<td>1.7K</td>
</tr>
<tr>
<td>Intermediate layer specifications</td>
<td>4.3K</td>
</tr>
<tr>
<td>Machine Model</td>
<td>1.8K</td>
</tr>
<tr>
<td>Correctness Proofs</td>
<td>20.1K</td>
</tr>
<tr>
<td>Security Proofs</td>
<td>4.8K</td>
</tr>
</tbody>
</table>
Inherit comprehensive virtualization features from KVM while protecting VM data

### Implementation

**Supported Functionality**

<table>
<thead>
<tr>
<th>Features verified secure</th>
<th>SeKVM</th>
<th>seL4</th>
<th>CertiKOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP hardware</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>SMP VMs</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Multiple VMs</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Shared page tables</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Virtio</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Device passthrough</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>VM migration</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Linux ease-of-use</td>
<td>✔</td>
<td>✕</td>
<td>✕</td>
</tr>
</tbody>
</table>
Performance Evaluation
Experimental Setup

- Measure network benchmarks from a bare metal client communicating with the server in the VM
- Evaluate on bare-metal/VM using 4-way SMP with 12 GB RAM with virtio
- VMs using end-to-end encrypted I/Os
- All workloads run on Arm server using Linux/KVM v4.18 based systems on Ubuntu 16.04

<table>
<thead>
<tr>
<th>Applications</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernbench</td>
<td>Kernel compile</td>
</tr>
<tr>
<td>Hackbench</td>
<td>Scheduler stress</td>
</tr>
<tr>
<td>Netperf</td>
<td>Network performance</td>
</tr>
<tr>
<td>Apache</td>
<td>Web server stress</td>
</tr>
<tr>
<td>Memcached</td>
<td>Key-val store</td>
</tr>
<tr>
<td>MySQL</td>
<td>Database workload</td>
</tr>
</tbody>
</table>
Performance Evaluation

Results

![Chart showing performance evaluation results for different benchmarks and applications. The chart compares KVM and SeKVM in terms of performance across various benchmarks and applications such as Kernbench, Hackbench, TCP_STREAM, TCP_MAERTS, TCP_RR, Apache, Memcached, and MySQL.]
Agenda

• A Secure and Formally Verified Commodity Multiprocessor Hypervisor
  • Retrofitting KVM
  • Verifying KVM
• Summary
• Future Work
Summary

• Verify KVM protects the confidentiality and integrity of VM data using microverification

• Introduced a layered hardware model that is simple to use for verification while accounting for realistic multiprocessor hardware features

• Our KVM implementation retains the commodity feature set and performance

• Our work is the first machine-checked security and correctness proof for a commodity multiprocessor hypervisor
Agenda

• A Secure and Formally Verified Commodity Multiprocessor Hypervisor
  • Retrofitting KVM
  • Verifying KVM

• Summary

• Future Work
Future Work

- Research opportunities for virtualization + security
  - What about availability guarantees for VMs? Protection for DoS attack?
  - Hardware support for VM security
- Formal verification requires uneasy effort
  - Program formally verified and secure software at the beginning
  - Scalable approach to replace or rewrite components of existing software with formally verified or secure code
Q&A - Thanks for listening!

shihwei@csie.ntu.edu.tw