Secure Guest Virtual Machine Support in Apparition

Ethan Johnson
Department of Computer Science
University of Rochester

In collaboration with Komail Dharsee and John Criswell
Two kinds of “VMs”

“Compiler-based VM”
- Restricts expressivity of architecture through virtual instruction set
- Enforces policy through instrumentation and run-time checks

Examples: CLR, Secure Virtual Architecture (Criswell et al., SOSP ‘07)

“Guest VM”
- A simulated full system
- Managed by a hypervisor
- Runs its own OS, apps independently

→ VMX supports these
Compiler-based VMs: Secure Virtual Architecture

- Virtual ISA for secure low-level software

- OS kernel in C/C++ compiles to extension of LLVM IR

- Special virtual instructions replace kernel assembly code
Protecting applications from the OS

- Compiler-based VM can enforce many policies

- *Virtual Ghost* lets user-space apps hide memory from kernel (Criswell et al., ASPLOS ’14)

- Performance overhead only on kernel mode, not user mode
Side-channel protections

- *Apparition* added side-channel protections to Virtual Ghost (Dong et al., Usenix Security ‘18)

- Prevents kernel, other apps from attacking ghost memory via:
  - Last-level-cache side channels
  - Page-fault side channels

*Apparition*
The *other* kind of VMs, in Apparition

- Want to run hypervisors under Apparition

- ...but VMX isn’t part of the V-ISA

- OK, so just extend the V-ISA. “How hard can it be?”
Introducing “Shade”

- Adds hardware virtualization support to Apparition
- Preserves protections for ghost memory on the host
- Mitigates side-channel attacks by guests and compromised kernel/hypervisor
# Shade architecture

<table>
<thead>
<tr>
<th>Host Applications (Native Code) (SVA V-ISA Code)</th>
<th>Guest Applications (Native Code)</th>
<th>Guest Applications (Native Code)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Guest OS (Native Code)</td>
<td>Guest OS (Native Code)</td>
</tr>
<tr>
<td></td>
<td>Hypervisor Driver</td>
<td></td>
</tr>
<tr>
<td>Host OS (SVA V-ISA Code)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shade Virtual Machine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Guest VMs

Virtual ISA

Native ISA
Key challenges we address

• Control flow integrity across VM entry/exit

• Hypervisor manages EPT but must not access ghost memory

• Over-powered guest could allow OS/hypervisor to escape Shade

• Side-channel mitigations
Control flow integrity on VM entry/exit

• VMX allows hypervisor to set arbitrary host state on exit
  • ...including RIP
  • ...and RSP, processor mode, segment registers...

• Easy for hypervisor to corrupt CFI, defeat enforcement

![Diagram showing normal and hijacked control flow](image-url)
Control flow integrity on VM entry/exit

• Shade must take control of VM entry/exit
  • Single virtual instruction for running a guest
    • Function call semantics
    • State saved/loaded from protected memory

• VMCS lives in ghost memory
  • Virtual instructions for reading and writing
  • Checks on values written

• Virtual instructions to access saved/loaded guest registers
Virtual instructions

• Allocate/free VMCS

• Load/unload VMCS onto processor

• Get/set guest registers managed by Shade

• Read/write VMCS fields

• Run VM guest
Protecting ghost memory

- Hypervisor must be able to add/remove EPT mappings
- But EPT could map protected memory into a guest
Protecting ghost memory

• Similar problem exists for host OS page table config

• Page tables stored in ghost memory

• Virtual instructions for MMU config
  • Shade tracks metadata on physical frame usage

• Checks prevent insecure EPT mappings
  • Ghost memory
  • Host page-table pages (regular + extended)
Virtual instructions for EPT

• Declare/undeclare PTP

• Update mapping

• Load root EPT pointer
Preventing over-powered guests

• VMX allows guests to run *native privileged* code
  • Not normally permitted in an SVA-based system
  • No opportunity to add instrumentation

• Guest effects on privileged state must be contained to guest

• Nothing host kernel not allowed to do should persist after VM exit
Preventing over-powered guests

- Some privileged state virtualized by hardware
  - CR3 with extended paging
  - Control registers saved.loaded atomically on entry/exit

- Other privileged state must be managed by hypervisor
  - Kernel MPX registers used by Shade for SFI
  - Shade must handle save/load during entry/exit

- Unused features can still be a threat
  - New processor features, MSRs
  - Shade checks VMCS writes to enforce safe defaults (VM exit)
Side-channel attacks

- Cache partitioning with Cache Allocation Technology (CAT)
  - Host kernel
  - Shade VM
  - Ghosting apps

- Must switch partition on VM entry/exit
  - VMs run in kernel/hypervisor partition
  - Possible to give each VM its own partition

- VMCS checks prevent guest access to CAT MSRs
Empirical evaluation
Benchmarks

• Extended Apparition prototype
  • FreeBSD 9.0 kernel ported to V-ISA
  • LLVM passes for SFI, CFI unmodified

• LMBench kernel latency benchmarks
  • Verify no new impact on host applications over Apparition

• Hypervisor microbenchmarks
  • Overheads of virtual instructions over native VMX operations
  • Hypothesis: hypervisor latency dominated by other factors
Host kernel benchmarks

- Kernel execution only; lower overhead for applications
- No new overheads over Apparition
Host kernel benchmarks - outliers

- Only affect ghosting applications
- No new overheads over Apparition
- Due to side-channel protections
  - Pre-allocating ghost memory affects fork()
  - Cache partitioning
Hypervisor microbenchmarks

Overheads should not noticeably affect hypervisor performance.
Future work

• Port full commodity hypervisor
  • FreeBSD’s bhyve
  • VirtualBox

• Protect guests from malicious hypervisor
  • Mitigate cloud compromise scenarios
  • VM exit handling, device virtualization pose challenges

• Prevent exploitation of hardware bugs
  • Processor bugs in VMX implementations
Summary

• Using VMX in existing Apparition system would compromise security

• We extend V-ISA to expose VMX in a “clean” way

• Minimal impact on hypervisor performance