Improving XMHF's Compatibility with Commodity Operating Systems and Hardware

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Xiaoyi Li

B.S., Computer Science and Engineering, University of California, Davis

Carnegie Mellon University
Pittsburgh, PA

May, 2023

© Xiaoyi Li, 2023
All Rights Reserved

Acknowledgements

First of all, I would like to thank my thesis advisor, Dr. Virgil Gligor, and reader, Dr. Miao Yu, for providing me with guidance throughout my research. When I focused too much on implementation details, they helped me maintain the big picture of the system.

I would also like to thank CyLab and ECE for providing the hardware used in my research. The different computers I have access to are essential for measuring the compatibility of XMHF+.

Thank you to Dr. Amit Vasudevan for the insightful discussions about XMHF. These discussions helped me to better understand the security properties and assumptions in the XMHF implementation. Additionally, thank you for taking the time to review my pull requests to XMHF's repository.

I would like to express my gratitude to the instructors of all computer science-related courses I took at UC Davis and CMU. The programming skills I learned in these classes are essential to the success of my research. In particular, I would like to thank Dr. Dave Eckhardt for teaching Operating System Design and Implementation (15-410/605). This class taught me how to efficiently find race conditions in programs, which directly helped me while debugging XMHF+.

I appreciate the contributions of open-source communities that provide high-quality software tools to people free of charge. My research relies on several open-source software, including QEMU, KVM, GDB, GCC, and Linux. I am also grateful to the software maintainers for taking the time to respond to the bug reports I submitted.

Finally, I would like to thank my family and friends for supporting me throughout

this research. This long journey would not have been possible without you.

This research is self-funded.

Abstract

Micro-hypervisors are used in many research projects to improve the security of computer systems. For example, some micro-hypervisors can separate security-sensitive programs from commodity operating systems, which typically consist of millions of lines of code. Thus, the security-sensitive programs are secure even if the operating systems are compromised. XMHF is a micro-hypervisor framework for the x86 micro-architecture that allows developers to extend it into customized micro-hypervisors. Unfortunately, XMHF does not support the latest commodity operating systems and hardware.

This thesis presents an enhancement of XMHF, called XMHF+, which addresses the compatibility issues mentioned above and introduces new features. XMHF+ extends its support to 64-bit modern operating systems such as Windows 10 and Debian 11, as well as modern chipsets with TPM 2.0. Moreover, XMHF+ virtualizes the hardware virtualization extension, enabling popular hypervisors such as KVM, VMware, VirtualBox, and Hyper-V to run on top of it. XMHF+ maintains the design principles of XMHF, making it possible to verify its memory integrity as future work.

Table of Contents

List of Tables

List of Figures

1

Introduction

Micro-hypervisors have been extensively used in recent research for security purposes [1–7]. Exemplary security properties include memory separation [1], execution environment separation [2,3], and I/O separation [4–7]. To minimize the engineering effort of building micro-hypervisors from scratch, XMHF [8] implements an extensible micro-hypervisor framework that encapsulates core functionalities of general micro-hypervisors and supports running a single rich guest operating system (OS). Other projects can extend XMHF with hypapps that implement custom security properties. The XMHF core consists of only around 6000 lines of code. The XMHF core is formally verified using the CBMC model checker to ensure that it protects the memory integrity of the micro-hypervisor (i.e., XMHF and its hypapp).

1.1 Problems

XMHF does not meet the requirements of modern x86 hardware and OSes in several ways. XMHF only supports 32-bit rich OSes, but not 64-bit ones. Additionally, XMHF does not support TPM 2.0, which is commonly found on new hardware. As a result, XMHF does not support modern operating systems such as Windows 10

and Debian 11.

XMHF does not support virtualization of the hardware virtualization extension, so rich OSes cannot utilize this feature of modern CPUs. As a result, rich applications that rely on general hypervisors like VirtualBox and Hyper-V cannot run on top of XMHF.

1.2 Challenges

In this thesis, we present XMHF+, which enhances XMHF by supporting modern OSes and hardware.

Improving XMHF to support new commodity OSes on new hardware is non-trivial. First, the micro-architectural requirements of modern OSes must be identified and fulfilled by XMHF+. For example, XMHF+ must virtualize the hardware virtualization extension in order to support modern general-purpose hypervisors.

Second, XMHF+ must maintain the security properties of XMHF. Although formal verification of XMHF+ is left as future work, XMHF+ must be designed to be verifiable. Therefore, XMHF+ must follow XMHF's design to minimize additional verification efforts while adding the necessary functionalities to support new OSes and hardware.

1.3 Contributions

We demonstrate that micro-hypervisors can be enhanced to support the latest OSes and hardware, while still isolating security-sensitive programs from untrusted OSes and applications. We present the design and implementation of XMHF+, which improves upon XMHF by adding support for modern 64-bit operating systems and Intel hardware. We implement virtualization of the hardware virtualization extension in XMHF+ to support general-purpose hypervisors. We show that the memory integrity of XMHF+ and its hypapp is verifiable, though its formal verification is

left as future work. TrustVisor [1] is a micro-hypervisor and is ported to run on top of XMHF [8]. In this thesis, we find and fix six security bugs in TrustVisor when running as a hypapp in XMHF.

The rest of this thesis is structured as follows. Section 2 discusses background information related to micro-hypervisors. Section 3 describes the design and implementation of XMHF+. Section 4 evaluates the new XMHF+ implementation based on code size, compatibility, and performance. Section 5 discusses the security vulnerabilities found in TrustVisor. Section 6 describes findings during XMHF+'s development and future work. Section 7 concludes the thesis.

2

Background

In this chapter, we discuss background information related to this thesis. We assume that the readers have basic knowledge about x86 systems programming.

2.1 Hardware Virtualization

Modern x86 CPUs are equipped with the hardware virtualization extension, which is commonly utilized by hypervisors to efficiently run virtual machines (VMs). Popular hypervisors include VMware, VirtualBox, KVM, and Hyper-V. In Intel CPUs, the hardware virtualization extension is implemented via virtual-machine extensions (VMX) [9]. In this thesis, we use the term "hardware virtualization extension" to refer to general x86 hardware virtualization technologies. We use the term "VMX" to refer specifically to hardware virtualization in Intel CPUs.

The hardware virtualization extension defines two modes for the CPU: host mode and guest mode. The hypervisor runs in host mode, and the virtual machines run in guest mode. When an event occurs in guest mode, the hardware generates an intercept and switches to host mode. In VMX, VM-entry refers to the switch from host mode to guest mode, which occurs when the hypervisor executes the launch virtual

machine (VMLAUNCH) and resume virtual machine (VMRESUME) instructions. VM-exit refers to the switch from guest mode to host mode, which occurs when an intercept is generated.

The hardware virtualization extension defines the virtual-machine control structure (VMCS), a data structure which transitions the host mode environment to the guest mode environment. In VMX, VMCS contains four types of fields. The guest-state fields specify the state of the guest after VM-entry. The host-state fields specify the state of the host after VM-exit. The control fields configure the CPU's behavior during guest mode. The information fields contain information about the last VM-exit.

The hardware virtualization extension defines the hardware page table (HPT) to translate memory addresses from guest mode to host mode. In VMX, HPT is implemented through the extended page table (EPT). The structure of EPT is similar to the page tables used by modern operating systems to support virtual memory. Similar to page tables, the hardware caches EPT entries in the translation lookaside buffer (TLB). Thus, the hypervisor must execute the invalidate translations derived from EPT (INVEPT) instruction to invalidate the TLB after changing or removing EPT entries.

2.1.1 Nested Virtualization

Nested virtualization refers to running a hypervisor inside another hypervisor. The hardware virtualization extension of x86 CPUs does not support nested virtualization in hardware. Thus, hypervisors supporting nested virtualization must virtualize the hardware virtualization extension. These hypervisors must also virtualize other hardware resources, such as I/O devices.

KVM [10] is a hypervisor that implements nested virtualization. KVM defines multiple levels of virtualization. KVM runs in L0, the guest hypervisors run in L1,

and guests of the L1 guest hypervisors run in L2. L0 runs in host mode, and all other levels run in guest mode.

To run an L2 guest, KVM must virtualize its VMCS. The L1 hypervisor provides VMCS12, which defines the transition between L1 and L2. KVM provides VMCS01, which defines the transition between L0 and L1. To implement nested virtualization, KVM must compute VMCS02, which defines the transition between L0 and L2.

KVM must virtualize intercepts from the L2 guest, which occur when events happen during L2 guest execution. If the event is related to KVM, KVM handles the intercept and resumes L2. Otherwise, KVM forwards the event to the L1 hypervisor by injecting the intercept into the L1 hypervisor.

KVM must also virtualize HPT. The L1 hypervisor provides HPT12, which translates L2 memory addresses to L1 memory addresses. KVM provides HPT01, which translates L1 memory addresses to L0 memory addresses. To run the L2 guest, KVM must build HPT02, which translates L2 memory addresses to L0 memory addresses. Initially, HPT02 is empty, and KVM builds it on-the-fly. When the L2 guest accesses a memory address that corresponds to an empty entry in HPT02 and triggers an HPT violation intercept, KVM computes the entry and adds it to HPT02. Whenever the L1 hypervisor modifies HPT12, KVM invalidates the corresponding entries in HPT02.

2.2 Micro-Hypervisor

Micro-hypervisors run underneath commodity operating systems and applications, and virtualize CPU and memory to achieve memory separation. Micro-hypervisors are fundamentally different from hypervisors because micro-hypervisors do not virtualize I/O devices. Due to their small size and limited features, micro-hypervisors have small trusted computing bases (TCBs) and simple designs. Thus, micro-hypervisors can be formally verified.

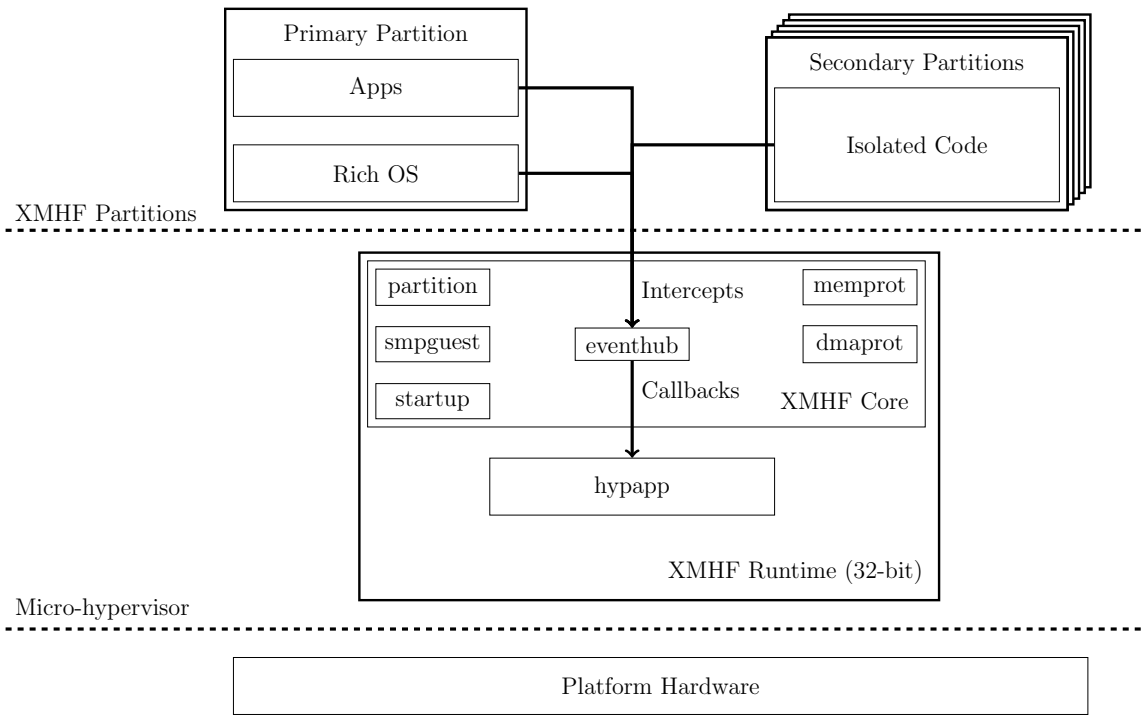

Figure 2.1: XMHF architecture, adapted from [8].

2.2.1 XMHF

XMHF [8] is a framework that assists the development of micro-hypervisors. The XMHF core implements the common logic in micro-hypervisors that interacts with the hardware virtualization extension. Developers can extend the functionality of XMHF through hypapps. Thus, different hypapps can share the same XMHF core, reducing development cost.

The architecture of XMHF is shown in Figure 2.1. XMHF supports only one full-featured rich OS and its applications running in the primary partition. The hypapps can create secondary partitions to run isolated code. XMHF provides APIs for hypapps to enforce memory separation between the primary partition and the secondary partitions.

The XMHF core has six components. The *startup* component performs initial-

ization after XMHF boots. The *smpguest* component supports running XMHF on multiple CPUs. The *partition* component initializes hardware virtualization extension data structures for XMHF partitions. The *eventhub* component handles all intercepts from XMHF partitions. The *memprot* component protects the memory integrity of XMHF and the hypapp from XMHF partitions using HPT. The *dmaprot* component protects the memory integrity of the XMHF and the hypapp from I/O devices through IOMMU page tables.

All intercepts generated by XMHF partitions are handled by the *eventhub* component. For intercepts related to the hypapp, as discussed below, the *eventhub* component calls the corresponding callback function in the hypapp to let the hypapp handle the intercept. The intercepts related to the hypapp are:

- The rich OS/Apps or isolated code accesses an I/O port reserved by the hypapp using the input from port (IN) or output to port (OUT) instruction.
- The rich OS/Apps or isolated code accesses invalid memory in the partition's HPT (i.e., HPT violation).
- The rich OS/Apps or isolated code invokes the hypapp using the call to VM monitor (VMCALL) instruction.
- The rich OS/Apps or isolated code tries to restart the machine.

Using the DRIVE methodology, [8] formally verifies that the XMHF core protects the memory integrity property of XMHF and the hypapp. A micro-hypervisor's memory integrity property can be formally-verified if the micro-hypervisor follows all the six properties required by the DRIVE methodology as discussed in Section 2.3.

2.2.2 TrustVisor

TrustVisor [1] is a micro-hypervisor that supports running security-sensitive programs that are separated from the rich OS/Apps. In TrustVisor, the security-

sensitive programs are called pieces of application logic (PALs). TrustVisor protects both the secrecy and integrity of PALs from the rich OS/Apps. [8] ports TrustVisor to run as a hypapp in XMHF.

The life cycle of a PAL consists of four types of events, as discussed below.

- **PAL Registration**: The rich OS/Apps register a PAL to TrustVisor, specifying the memory regions in which the PAL operates, the argument format for calling the PAL, and the entry point of the PAL. TrustVisor moves the PAL's memory from the XMHF primary partition to a newly-created XMHF secondary partition. This ensures memory separation between the rich OS/Apps and the PAL.

- **PAL Invocation**: The rich OS/Apps call the entry point of the PAL using a C function call. TrustVisor copies all arguments from the rich OS/Apps to the PAL. Then, TrustVisor executes the PAL. While the PAL is executing, interrupts are disabled to ensure that the PAL executes in a single-threaded environment.

- **PAL Termination**: The PAL completes its execution and executes a C function return. TrustVisor copies the return value from the PAL to the rich OS/Apps. Then, TrustVisor resumes the rich OS/Apps.

- **PAL Unregistration**: The rich OS/Apps unregister a PAL from TrustVisor. TrustVisor clears the PAL's memory, which may contain secret information, and then moves the PAL's memory from the XMHF secondary partition back to the XMHF primary partition.

2.3 Security Properties Required by the DRIVE Methodology

The DRIVE design methodology [8] enables formal verification of memory integrity of micro-hypervisors. DRIVE defines six properties that micro-hypervisors must follow. XMHF follows these properties, and it assumes its hypapp also follows these properties. Thus, the memory integrity of XMHF and its hypapp can be formally verified.

2.3.1 Modularity

The modularity property (MOD) consists of two sub-properties. First, when the micro-hypervisor initializes, its $init()$ function needs to be called. Second, when an intercept is triggered, the hardware needs to transfer control to one of the micro-hypervisor's intercept handlers: $ih_1(), ..., ih_k()$.

2.3.2 Atomicity

The atomicity property (ATOM) consists of two sub-properties. ATOM_{init} requires that $init()$ is executed at the start of micro-hypervisor initialization in a single-threaded environment. ATOM_{ih} requires that when an intercept handler is running, no other intercept handlers or rich OS/Apps may run.

2.3.3 Memory Access Control Protection

The memory access control protection property (MPROT) mandates the use of a memory access control mechanism ($MacM$) stored in the micro-hypervisor's memory. $MacM$ consists of two hardware access control components: $MacM_G$, which controls memory accesses of the rich OS/Apps, and $MacM_D$, which controls memory accesses of devices.

2.3.4 Correct Initialization

The correct initialization property (INIT) has two requirements. First, after the micro-hypervisor is initialized, $MacM$ must protect the micro-hypervisor's memory from the rich OS/Apps and devices. Second, the micro-hypervisor must initialize the intercept entry points to point to the correct intercept handlers.

2.3.5 Proper Mediation

The proper mediation property (MED) requires that $MacM$ is active whenever untrusted programs or devices may access memory. First, $MacM_G$ must be active when any code in the rich OS/Apps executes. Second, $MacM_D$ must always be active, because devices can access memory at any time.

2.3.6 Safe State Updates

The safe state updates property (SAFEUPD) requires that when updating the micro-hypervisor's memory and hardware control structures, intercept handlers must ensure that (1) $MacM$ protects the micro-hypervisor's memory, (2) intercept entry points are not modified, and (3) the micro-hypervisor's code is not modified.

2.4 Related Work

Lockdown [2] is a micro-hypervisor that implements separation between two commodity operating systems. Similar to TrustVisor, it is ported to run on top of XMHF as an example hypapp [8]. Compared to TrustVisor, Lockdown needs to separate more resources, such as I/O devices. Lockdown performs slower and less frequent switches between the trusted and untrusted environments than TrustVisor.

SecVisor [11] is a micro-hypervisor that ensures the code integrity of the rich OS. It uses HPT to ensure that only approved code can execute in kernel mode of the

rich OS/Apps. Unlike TrustVisor and Lockdown, SecVisor focuses solely on integrity and does not address secrecy.

Wimpy kernel [4] is a micro-kernel that implements on-demand I/O isolation. Custom security-sensitive programs that require isolated I/O can run on the wimpy kernel. This project uses XMHF to implement a micro-hypervisor that separates the wimpy kernel from the rich OS/Apps.

[12] redesigns XMHF into üXMHF. üXMHF supports multiple hypapps, while XMHF supports only one hypapp. üXMHF follows a different approach toward formal verification compared to XMHF. üXMHF uses Frama-C to formally verify multiple security properties, including control-flow integrity and memory integrity. In contrast, XMHF uses CBMC to verify only the memory integrity property.

CloudVisor [13] is a micro-hypervisor that virtualizes the hardware virtualization extension. It runs a commodity hypervisor in guest mode and protects the security of the guest VMs of the commodity hypervisor. For example, CloudVisor prevents the commodity hypervisor from accessing secrets in its guest VMs. In contrast, XMHF+ does not provide additional security to the guest VMs.

Bareflank [14] is a micro-hypervisor framework written in C++ that enables developers to implement custom micro-hypervisor logic in an extension. It satisfies modern OS compatibility requirements, including 64-bit support and Unified Extensible Firmware Interface (UEFI) booting. Unlike XMHF and XMHF+, Bareflank supports running multiple rich OSes natively. However, as far as we know, Bareflank is not designed to be formally verified.

3

Design and Implementation

In this chapter we discuss the security model of XMHF+. Then we discuss the XMHF+ design, which achieves compatibility and security. Finally, we discuss implementation details of XMHF+.

3.1 Security Model

XMHF+ has a similar security model to XMHF [8]. The goal of XMHF+ is to protect memory integrity of the micro-hypervisor (i.e., XMHF+ and its hypapp).

XMHF+ has the same threat model as XMHF. The attacker can control the rich OS/Apps and I/O devices, and run his/her own isolated code. XMHF+ assumes that the attacker is remote, i.e., does not have physical access to the hardware. An attacker can attempt to attack memory integrity of the micro-hypervisor by (1) attacking XMHF+'s initialization, (2) accessing memory from the rich OS/Apps or isolated code, (3) accessing memory from I/O devices, and (4) generating intercepts that will be handled by XMHF+.

Similar to XMHF, XMHF+ assumes the control flow integrity (CFI) of XMHF+. However, future work is required to prove and enforce this property.

XMHF+ assumes that the hardware and firmware are trusted. We consider hardware vulnerabilities like Spectre [15] and Meltdown [16] as out of scope. XMHF+ also does not protect against denial-of-service attacks.

3.2 Design

3.2.1 Compatibility

Hardware resources consist of CPU, memory, and I/O devices. XMHF+ must ensure that sharing hardware resources between the rich OS/Apps and XMHF+ preserves compatibility with the rich OS/Apps.[1] For any hardware resource, XMHF+ must authorize the rich OS/Apps to access the resource with exactly one of the following three policies.

- **Exporting**: For any hardware resource that is not required by XMHF+, XMHF+ must export the resource to the rich OS/Apps. This means that they are always allowed to access the hardware resource directly. Note that XMHF+ exports all I/O devices to the rich OS/Apps because XMHF+ is a micro-hypervisor and does not perform device virtualization.

- **Hiding**: For any hardware resource required by XMHF+ and not required by the rich OS/Apps, XMHF+ can hide the resource from them by dropping their access to the resource and reporting that the resource is unavailable. This is because well-behaved rich OS/Apps will then stop using the hardware resource and continue to function properly. However, hiding the resource may break the compatibility of future rich OS/Apps that require the resource. For example, XMHF+ hides the IOMMU from the rich OS/Apps because they can

[1] The hypapp ensures that sharing hardware resources between the rich OS/Apps and the isolated code preserves compatibility with the rich OS/Apps.

run without the IOMMU. Virtualization of the IOMMU in XMHF+ is left as future work.

- **Virtualizing**: For any hardware resource required by both XMHF+ and the rich OS/Apps, XMHF+ must virtualize the resource, ensure the resource virtualization does not violate its security properties, and emulate the resource for all secure accesses. For example, XMHF+ virtualizes non-maskable interrupts (NMIs) because XMHF+ implements quiescing through NMIs, and the rich OS/Apps require NMIs.

To make XMHF+ compatible with rich OS/Apps, we first identify all hardware resources required by XMHF+. For each resource, we decide whether to virtualize or hide the resource. Then we export all hardware resources not required by XMHF+ to the rich OS/Apps.

3.2.2 Virtualizing the Hardware Virtualization Extension

XMHF+ relies on the hardware virtualization extension to virtualize CPU and memory. However, modern rich OS/Apps also require the hardware virtualization extension to run hypervisors. Therefore, XMHF+ must virtualize the hardware virtualization extension to support modern rich OS/Apps.

KVM [10] implements nested virtualization, which includes virtualizing the hardware virtualization extension. XMHF+ borrows several ideas from this work. However, XMHF+ does not implement nested virtualization since it does not virtualize I/O devices.

We modify the architecture of XMHF to virtualize the hardware virtualization extension in XMHF+. The modified XMHF+ architecture is shown in Figure 3.1. The primary partition can run in two modes: guest VMs of the general-purpose hypervisor run in L2, and the rich OS/Apps and the general-purpose hypervisor run

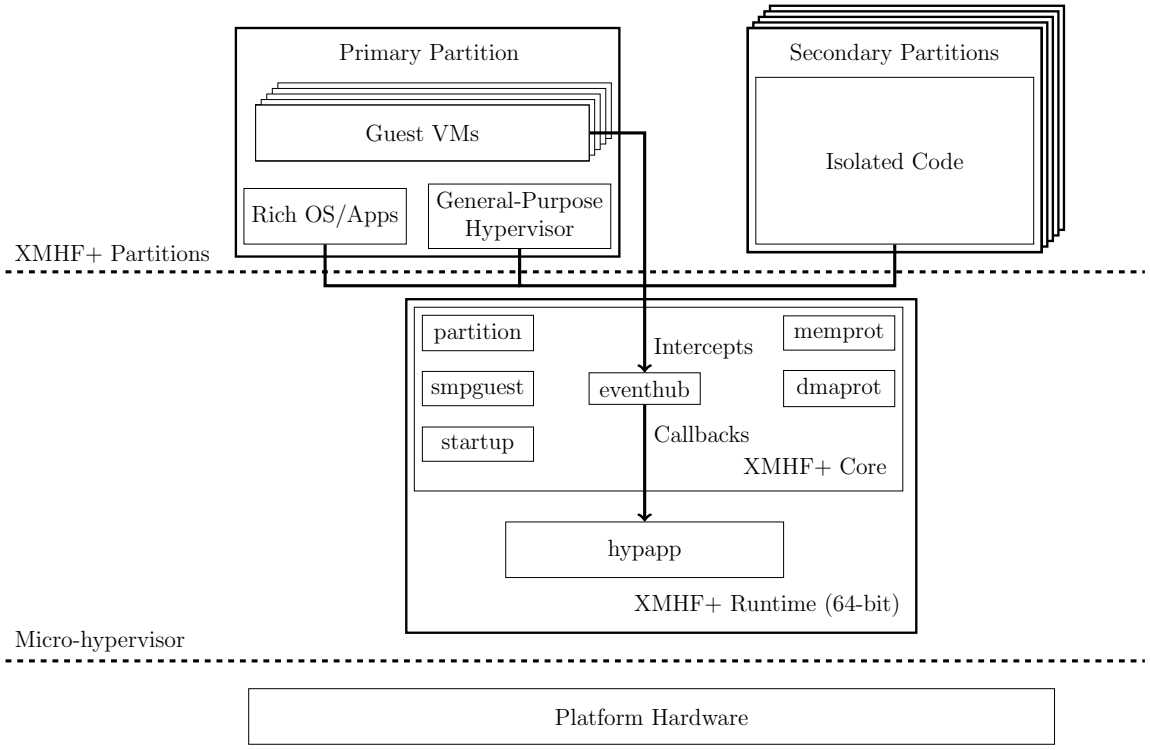

Figure 3.1: XMHF+ architecture.

in mL1. XMHF+ runs in mL0. While mL1 and mL0 are superficially similar to L1 and L0 in nested virtualization [10], they are actually different because XMHF+ does not need to virtualize devices. Moreover, the secondary partitions created by hypapps do not need a notion of L2 or mL1 because these hypapps must have low complexity to support their security goals. Specific hypapps can define L2 and mL1 for secondary partitions at the cost of increasing hypapp complexity.

Virtualizing VMCS

To run the guest VMs, the general-purpose hypervisor provides VMCS12, which defines execution environment transition between the general-purpose hypervisor and the guest VMs. XMHF+ must translate VMCS12 to VMCS02, which defines the transition between XMHF+ and the guest VMs.

Translating VMCS12 to VMCS02 depends on the hardware resource represented by each VMCS field. If XMHF+ exports the resource to the rich OS/Apps, the field can be directly copied from VMCS12 to VMCS02. If XMHF+ hides the resource from the rich OS/Apps, XMHF+ needs to modify the field in VMCS02 to prevent the guest VMs from accessing the resource. If XMHF+ virtualizes the resource, the specific virtualization logic computes the VMCS02 value from VMCS12.

Virtualizing Intercepts

The hardware virtualization extension generates intercepts when events occur while the rich OS/Apps, the general-purpose hypervisor, and the guest VMs are running. To virtualize the hardware virtualization extension, XMHF+ must handle intercepts from the guest VMs correctly. Intercepts from the guest VMs are handled differently from intercepts from the rich OS/Apps and the general-purpose hypervisor.

Similar to KVM [10], XMHF+ handles each intercept from the guest VMs in one of two ways, depending on the reason of the intercept. If the intercept is caused by an event related to XMHF+, XMHF+ handles the event and resumes executing the guest VMs. Otherwise, XMHF+ forwards the event to the general-purpose hypervisor. This is done by injecting the intercept into the general-purpose hypervisor.

Some intercepts from the rich OS/Apps cause XMHF+ to call callback functions in the hypapp, such as HPT violation and I/O port access. When the guest VMs generate these intercepts, XMHF+ also calls the same callback functions in the hypapp. Hence, the hypapp's callback functions must be updated to handle callbacks from both the rich OS/Apps and the guest VMs correctly. For example, the hypapp must implement different logics when accessing memory of the rich OS/Apps and the guest VMs. The hypapp can access memory of the rich OS/Apps directly, but the memory of the guest VMs can only be accessed after performing the address translation defined by the general-purpose hypervisor.

Virtualizing HPT

XMHF+ uses HPT to enforce memory integrity of the micro-hypervisor. However, modern hypervisors also require HPT to implement memory virtualization. Thus, XMHF+ must virtualize HPT to support modern hypervisors.

In XMHF+, HPT01 protects memory integrity of the micro-hypervisor from the rich OS/Apps. The general-purpose hypervisor specifies HPT12, which translates guest VM memory addresses to rich OS/Apps memory addresses. To run the guest VMs, XMHF+ builds HPT02, which translates guest VM memory addresses to XMHF+ memory addresses. HPT02 contains the same entries as HPT12, except for the entries that violate memory integrity of the micro-hypervisor. Thus, guest VMs cannot violate memory integrity of the micro-hypervisor.

Similar to KVM [10], XMHF+ builds HPT02 on demand. Initially, HPT02 contains no entries. When the guest VMs access memory not in HPT02, XMHF+ receives an intercept. XMHF+ checks whether the access violates the micro-hypervisor's memory integrity. If the access is authorized, XMHF+ computes the HPT02 entry from HPT12 and resumes the guest VMs.

3.2.3 Verifiability

As discussed in Section 2.3, memory integrity of XMHF and its hypapp is formally verified using the DRIVE methodology [8]. To ensure verifiability, XMHF+ also follows all properties required by DRIVE. Furthermore, XMHF+ aims to minimize the virtualization of hardware resources to reduce complexity and facilitate formal verification. It should be noted that formal verification of XMHF+ is left as future work.

Modularity

XMHF+ preserves both sub-properties of the modularity property (MOD). First, XMHF+ maintains the initialization logic of XMHF. During dynamic root of trust measurement (DRTM), the CPU transfers control to XMHF+'s *init*() function in the same way as it does for XMHF. Second, XMHF+ uses the same intercept handling approach as XMHF. XMHF+ only modifies existing intercept handlers and adds new ones as required. When an intercept is triggered, the hardware still transfers control to one of XMHF+'s intercept handlers.

Atomicity

XMHF+ preserves both sub-properties of the atomicity property (ATOM), i.e., ATOM_{init} and ATOM_{ih}. Similar to XMHF, XMHF+ utilizes hardware support (i.e., DRTM) to ensure atomicity during boot. The hardware ensures that *init*() is called with only one CPU running and interrupts disabled, thus preserving ATOM_{init}.

XMHF+ preserves ATOM_{ih} through a similar quiescing mechanism as XMHF. Each intercept handler can execute independently without interaction with other intercept handlers or partitions. XMHF+ follows the quiesce implementation of XMHF where each CPU quiesces all other CPUs before running the intercept handler. The other CPUs enter a busy wait loop until the first CPU completes the intercept handler. This mechanism preserves ATOM_{ih} because while one CPU is executing the intercept handler, all other CPUs are in the busy wait loop.

Memory Access Control Protection

XMHF+ satisfies the memory access control protection property (MPROT). Similar to XMHF, XMHF+ enforces the micro-hypervisor's memory integrity through the $MacM$ mechanism. $MacM$ consists of two components: $MacM_G$ and $MacM_D$, which respectively control memory accesses of XMHF+ partitions and devices.

XMHF+ extends $MacM_G$ to support running guest VMs. In XMHF, HPT always protects the micro-hypervisor's memory from the rich OS/Apps. In XMHF+, HPT01 and HPT02 respectively always protect the micro-hypervisor's memory from the rich OS/Apps and the guest VMs. Both HPT01 and HPT02 are stored in XMHF+ memory. The implementation of $MacM_D$ in XMHF+ remains the same as in XMHF because XMHF+ does not deal with device virtualization.

Correct Initialization

XMHF+ satisfies the correct initialization property (INIT). XMHF+ does not modify the initialization code for intercept handlers in XMHF, so the INIT property of XMHF ensures that intercept entry points point to the correct intercept handlers. The initialization logic from XMHF also ensures that HPT01 protects the micro-hypervisor's memory after initialization. XMHF+ initializes HPT02 to reject all memory accesses from guest VMs, which means that HPT02 initially protects the micro-hypervisor's memory. Therefore, $MacM_G$ protects the micro-hypervisor's memory. The initialization logic from XMHF also ensures that $MacM_D$ protects the micro-hypervisor's memory after initialization.

Proper Mediation

XMHF+ satisfies the proper mediation property (MED). When guest VMs are running, XMHF+ ensures that HPT02 is active. When the rich OS/Apps and the general-purpose hypervisor are running, XMHF+ ensures that HPT01 is active. Thus, $MacM_G$ is always active when the XMHF+ primary partition is running. The logic from XMHF also ensures that $MacM_G$ is active when XMHF+ secondary partitions are running.

Same as XMHF, XMHF+ activates $MacM_D$ during initialization and never deactivates it. Thus, the memory accesses of devices are always controlled by $MacM_D$.

Safe State Updates

XMHF+ satisfies all three requirements of the safe state updates property (SafeUpd). As mentioned in Section 2.3.6, intercept handlers in XMHF+ must ensure that (1) $MacM$ protects the micro-hypervisor's memory, (2) intercept entry points are not modified, and (3) XMHF+'s code is not modified.

For (1), XMHF+ follows the well-defined interfaces provided by XMHF when modifying HPT01. These interfaces ensure that HPT01 always protects the micro-hypervisor's memory. XMHF+ only modifies HPT02 by constructing its entries from HPT01. As a result, XMHF+ guarantees that the memory addresses in HPT02 are always a subset of HPT01. Thus, $MacM_G$ always protects the micro-hypervisor's memory. XMHF+ also follows the well-defined interfaces in XMHF when modifying $MacM_D$.

For (2) and (3), XMHF+ intercept handlers never modify intercept entry points or XMHF+ code. Thus, the Safe State Updates property is preserved.

3.3 Implementation

3.3.1 64-bit Support

XMHF statically maps all 4 GiB of memory that can be accessed by a 32-bit rich OS/Apps. However, XMHF+ cannot map all possible 64-bit memory addresses due to the large address space. Therefore, XMHF+ requires configuration of the maximum memory address at compile time. XMHF+ maps all memory below this configured maximum memory address.

XMHF+ must support both 32-bit and 64-bit rich OS/Apps [9]. Thus, XMHF+ implements new APIs for hypapps to query the mode of the running rich OS/Apps. These new APIs allow hypapps to distinguish between 32-bit and 64-bit rich OS/Apps and take appropriate action. For example, TrustVisor uses these APIs to use

different calling conventions for 32-bit PALs and 64-bit PALs.

3.3.2 Compatibility

As discussed in Section 3.2.1, for each hardware resource, XMHF+ determines the policy for rich OS/Apps to access the resource with one of the following three policies: exporting, hiding, and virtualizing.

CPU

For CPU instructions, XMHF+ virtualizes all VMX instructions, as well as the read from model specific register (RDMSR), write to model specific register (WRMSR), and CPU identification (CPUID) instructions. XMHF+ hides all Safer Mode Extensions (SMX) instructions. All other instructions are exported to the rich OS/Apps.

- XMHF+ virtualizes VMX instructions. XMHF+ uses VMX to enforce memory integrity of the micro-hypervisor, and rich OS/Apps use VMX to run general-purpose hypervisors.
- XMHF+ virtualizes the RDMSR and WRMSR instructions to virtualize model specific registers (MSRs).
- XMHF+ virtualizes the CPUID instruction to hide hardware resources from the rich OS/Apps. The rich OS/Apps detect the availability of many hardware resources through the CPUID instruction. When the rich OS/Apps execute the CPUID instruction, VMX generates an intercept, and XMHF+ returns a modified CPUID result that hides hardware resources from the rich OS/Apps.
- XMHF+ hides SMX instructions (i.e., GETSEC). SMX instructions are used to perform DRTM, and rich OS/Apps do not require DRTM.
- One compatibility bug in XMHF is due to not handling the invalidate process-context identifier (INVPCID), read time-stamp counter and processor ID (RDTSCP),

and restore processor extended states supervisor (XRSTORS) instructions correctly. The rich OS/Apps can detect the availability of these instructions through CPUID. XMHF does not change the result of CPUID, but XMHF disables these instructions in the VMX configuration. Thus, the rich OS/Apps crashes when accessing these instructions. XMHF+ corrects the VMX configuration to export these instructions to the rich OS/Apps.

For CPU registers, XMHF virtualizes a subset of MSRs, as discussed below. All other CPU registers are exported to the rich OS/Apps, including all general-purpose registers, control registers, floating-point unit (FPU) registers, single instruction, multiple data (SIMD) registers, and all other MSRs.

- XMHF+ virtualizes the memory type range registers (MTRR) MSRs. XMHF+ uses MTRRs to control memory caching when accessing memory. Thus, XMHF+ virtualizes MTRRs and uses EPT memory type bits to control memory caching when the rich OS/Apps access memory. One compatibility bug in XMHF occurs because XMHF does not virtualize MTRRs. Instead, XMHF exports MTRRs to the rich OS/Apps. As a result, XMHF crashes when a modern rich OS accesses memory-mapped I/O with incorrect caching.

- XMHF+ virtualizes the extended feature enable register (EFER) MSR. XMHF+ requires EFER to run correctly in 64-bit mode. The rich OS/Apps also requires EFER to run in 64-bit mode. Thus, XMHF+ virtualizes EFER.

- XMHF+ virtualizes the `BIOS_UPDT_TRIG` MSR, which controls CPU microcode update. XMHF+ must verify the CPU microcode update to prevent the rich OS/Apps from loading malicious CPU microcode updates to the CPU.

For CPU interrupts, XMHF+ only virtualizes NMIs. XMHF+ requires NMIs to implement quiescing, but rich OS/Apps require NMIs to handle non-recoverable

errors from the hardware. XMHF+ exports all maskable interrupts to the rich OS-/Apps.

For the Advanced Programmable Interrupt Controller (APIC), XMHF+ only virtualizes the interrupt command register (ICR) register in APIC. The rich OS/Apps require the ICR to send interrupts between CPUs, and XMHF+ uses the ICR to send NMIs to other CPUs to implement quiescing. All other registers in APIC are exported to the rich OS/Apps.

For the input-output memory management unit (IOMMU), XMHF+ hides it from the rich OS/Apps. XMHF+ requires the IOMMU to protect the micro-hypervisor's memory from devices. XMHF+ modifies the Advanced Configuration and Power Interface (ACPI) table to prevent the rich OS/Apps from detecting the IOMMU. XMHF+ also removes the memory addresses of the IOMMU from EPT to prevent malicious rich OS/Apps from accessing it. However, hiding the IOMMU may break the compatibility of future rich OS/Apps, so virtualization of the IOMMU is left as future work.

Memory

Same as XMHF, XMHF+ separates memory into two regions: one for the micro-hypervisor, and one for XMHF+ partitions. The first region is hidden from the rich OS/Apps, and the second region is exported to the rich OS/Apps. During XMHF+ initialization, EPT is configured to protect the micro-hypervisor's memory from the rich OS/Apps.

Operating systems booted by BIOS use E820 to detect memory, as specified by ACPI [17]. Same as XMHF, XMHF+ modifies E820 records to hide the micro-hypervisor's memory. Thus, benign rich OS/Apps do not attempt to access the micro-hypervisor's memory.

Operating systems booted by UEFI use the `GetMemoryMap` function to detect

memory. The support for UEFI in XMHF+ is left as future work. We plan to hide the micro-hypervisor's memory by providing a wrapper function to rich OS/Apps booted by UEFI. The wrapper function calls `GetMemoryMap` and modifies the result to hide the micro-hypervisor's memory.

I/O Devices

Same as XMHF, XMHF+ exports all I/O devices to the rich OS/Apps. XMHF+ configures EPT to export all memory-mapped I/O to the rich OS/Apps. XMHF+ configures VMCS to export all I/O ports to the rich OS/Apps. Thus, as a micro-hypervisor, XMHF+ does not perform any device virtualization.

3.3.3 Virtualizing the Hardware Virtualization Extension

As discussed in Section 3.2.2, to support running guest VMs, XMHF+ must virtualize VMCS, intercepts, and HPT.

Virtualizing VMCS

The general-purpose hypervisor defines the execution environment transition between the general-purpose hypervisor and the guest VMs through VMCS12. XMHF+ translates all fields in VMCS12 to VMCS02, which defines the transition between XMHF+ and the guest VMs.

In VMX, VMCS contains four types of fields, as mentioned in Section 2.1: guest-state fields, host-state fields, control fields, and information fields. XMHF+ handles each type of fields differently when virtualizing VMCS.

The guest-state fields of VMCS12 define the state of the guest VMs. XMHF+ copies all the guest-state fields to VMCS02.

The host-state fields of VMCS12 define the state of the general-purpose hypervisor. However, when an intercept happens, VMX must execute the intercept handler

of XMHF+. Thus, XMHF+ replaces all host-state fields with the state of XMHF+ in VMCS02.

The control fields of VMCS12 define the behavior of the CPU when the guest VMs are running. XMHF+ hides and virtualizes some VMX control fields, as discussed below. All other control fields are directly copied to VMCS02.

- The MSR load and store control fields and the MSR bitmap control field are virtualized. Both XMHF+ and the general-purpose hypervisor require these control fields to virtualize MSRs.

- The EPT pointer (EPTP) and virtual-processor identifier (VPID) control fields are virtualized. XMHF+ requires EPTP and VPID to enforce the memory integrity of XMHF+ and its hypapp. The general-purpose hypervisor also requires EPTP and VPID to virtualize memory.

- The accessed and dirty flags for EPT control field is hidden from the general-purpose hypervisor. This control field cannot be virtualized under the way XMHF+ virtualizes EPT. Nevertheless, the general-purpose hypervisor does not require this control field.

- The control fields that specify memory addresses of the general-purpose hypervisor are virtualized. XMHF+ must check these addresses to ensure the memory integrity of the micro-hypervisor. If these addresses do not violate the memory integrity of the micro-hypervisor, XMHF+ directly copies the content of the control fields from VMCS12 to VMCS02.

- VM-execution control bits related to NMI and the VM-entry interruption-information field are virtualized. XMHF+ requires these fields to virtualize NMI interrupts when the guest VMs are running. The VM-execution control bits related to NMI allow XMHF+ to receive all NMIs. The VM-entry

interruption-information field allows XMHF+ to inject NMIs into the guest VMs.

When an intercept from the guest VMs occurs, the hardware writes information related to the intercept to the information fields of VMCS02. As discussed below, if XMHF+ decides to inject the intercept into the general-purpose hypervisor, XMHF+ copies the information fields of VMCS02 to VMCS12.

Virtualizing Intercepts

How XMHF+ handles an intercept from the guest VMs depends on the event that causes the intercept. The following events in guest VMs are related to XMHF+, so XMHF+ handles intercepts caused by these events differently. Other events are not related to XMHF+, so XMHF+ simply forwards these events to the general-purpose hypervisor.

- EPT violation is related to XMHF+ because XMHF+ virtualizes EPT.
- NMI interrupt and NMI window are related to XMHF+ because XMHF+ virtualizes NMIs. The NMI interrupt intercept allows XMHF+ to handle all NMIs received by the CPU. The NMI window intercept allows XMHF+ to correctly inject NMIs into the rich OS/Apps.
- Execution of the CPUID, IN, OUT, and VMCALL instructions is related to XMHF+ because these instructions are part of the interface between XMHF+ partitions and hypapps. XMHF+ virtualizes these intercepts so that the guest VMs can detect the presence of the hypapp through CPUID and interact with the hypapp through IN, OUT, and VMCALL.
- Execution of the WRMSR and RDMSR instructions is related to XMHF+ because XMHF+ virtualizes the MSR bitmap.

Virtualizing HPT

In VMX, HPT is implemented through EPT. XMHF+ uses EPT01 to enforce memory integrity of the micro-hypervisor. When the rich OS/Apps and the general-purpose hypervisor are running, EPT does not need to be virtualized. Thus, XMHF+ directly loads EPT01 to VMX.

When guest VMs are running, XMHF+ must virtualize EPT. XMHF+ requires EPT to enforce memory integrity of the micro-hypervisor, but the general-purpose hypervisor also requires EPT to virtualize memory. The general-purpose hypervisor provides XMHF+ with EPT12, which translates guest VM memory addresses to rich OS/Apps memory addresses. XMHF+ then translates EPT12 to EPT02 by removing memory addresses that cannot be accessed through EPT01. EPT02 is provided to VMX to execute the guest VMs. Thus, XMHF+ guarantees that the memory addresses accessible through EPT02 are always a subset of the memory addresses accessible through EPT01, ensuring that the guest VMs do not violate memory integrity of the micro-hypervisor.

Similar to KVM [10], XMHF+ builds EPT02 on-the-fly. Initially, EPT02 is empty when a guest VM is launched. When the guest VM accesses memory that corresponds to an empty entry in EPT02, it triggers an EPT violation intercept. In the EPT violation intercept handler, XMHF+ checks the corresponding entry in EPT12. If the entry in EPT12 is valid and translates the guest VM address to a general-purpose hypervisor address, XMHF+ computes the empty entry in EPT02 using the general-purpose hypervisor address and retries the instruction in the guest VM that causes the EPT violation. However, if the entry in EPT12 is invalid, XMHF+ forwards the EPT violation to the general-purpose hypervisor.

However, unlike KVM, XMHF+ does not track changes in EPT12 and update EPT02 dynamically. Instead, when the general-purpose hypervisor executes the

INVEPT instruction, XMHF+ removes all entries in EPT02. The VMX specification requires the general-purpose hypervisor to execute the INVEPT instruction after a valid entry in EPT12 is changed or removed. Although removing all entries in EPT02 when the general-purpose hypervisor executes INVEPT results in high run time overhead for XMHF+, it avoids communication between intercept handlers of different CPUs and enables quiescing.

Updating TrustVisor

TrustVisor must be updated to support launching PALs from the guest VMs. To enable this, we make the following changes to TrustVisor.

- TrustVisor must distinguish between EPT01 and EPT02. TrustVisor must use EPT02 to read or write guest VMs' memory and use EPT01 to add or remove memory from the primary partition. In contrast, when PALs are launched from the rich OS/Apps, EPT01 is always used.

- TrustVisor must ensure that when PALs are running, asynchronous events do not cause intercepts that are forwarded to the general-purpose hypervisor. For example, the general-purpose hypervisor may configure VMCS12 to generate intercepts when interrupts arrive at the CPU. Thus, TrustVisor must modify VMCS02 to block all interrupts when PALs are running.

A limitation of supporting launching PALs from the guest VMs is related to memory locking. TrustVisor requires that the memory of PALs must not be swapped. Before a PAL is registered by the rich OS/Apps, the `mlock` system call is used to prevent the rich OS from swapping the PAL's memory. However, when a PAL is registered by the guest VMs, there is no standard way to prevent the general-purpose hypervisor from swapping the PAL's memory. Currently, TrustVisor halts when this situation occurs, as TrustVisor's memory separation policy does not allow

the general-purpose hypervisor, which is located in the XMHF+ primary partition, to access the memory of the PAL, which is located in an XMHF+ secondary partition.

3.3.4 Virtualizing NMIs

Non-maskable interrupts (NMIs) are required by both XMHF+ and the rich OS/Apps, so XMHF+ must virtualize NMIs. XMHF+ uses NMIs to implement quiescing. The rich OS/Apps require NMIs to receive notification on non-recoverable hardware errors. Linux also implements watchdog using NMI [18].

NMIs can arrive at the CPU at any time when XMHF+ and rich OS/Apps are running. If an NMI arrives when XMHF+ is running, the CPU invokes the NMI interrupt handler in XMHF+.[2] If an NMI arrives when the rich OS/Apps are running, the CPU invokes the NMI VM-exit handler[3] in XMHF+.

Both the NMI interrupt handler and NMI VM-exit handler in XMHF+ implement the same NMI virtualization logic. This NMI virtualization logic checks whether another CPU is requesting quiescing. If so, it invokes the logic for quiescing, which involves entering a busy wait loop until the other CPU ends the quiescing. If not, it modifies VMCS to inject the NMI into the rich OS/Apps.

To prevent the NMI interrupt handler or NMI VM-exit handler from being interrupted by another NMI, Intel defines the behavior of NMI blocking. NMIs become blocked at the start of the NMI interrupt handler and NMI VM-exit handler. NMIs become unblocked when the CPU executes the interrupt return (IRET) instruction. The CPU only invokes the NMI interrupt handler or NMI VM-exit handler when

[2] The NMI interrupt handler breaks the atomicity of the XMHF+ code that the NMI interrupts. Thus, both XMHF and XMHF+ require manual auditing of the NMI interrupt handler during formal verification.

[3] In this section, we use "VM-exit" instead of "intercept" to prevent confusion between "NMI interrupt handler" and "NMI intercept handler". The NMI interrupt handler in XMHF+ is not one of the NMI VM-exit handlers (i.e., NMI intercept handlers) in XMHF+. The former is an interrupt handler in x86, while the latter is a VM-exit handler in VMX.

NMIs are blocked [9]. Thus, the NMI interrupt handlers in XMHF+ and rich OS-/Apps do not need to be reentrant.

Injecting NMIs into the Rich OS/Apps

One challenge of implementing NMI virtualization is that XMHF+ must respect NMI blocking of the rich OS/Apps. After XMHF+ injects an NMI into the rich OS/Apps, it must wait until the rich OS/Apps unblock NMIs before injecting another NMI into the rich OS/Apps.

To correctly inject NMIs into the rich OS/Apps, XMHF+ enables virtual NMIs in the VMX configuration. With virtual NMIs enabled, VMX monitors whether the rich OS/Apps are blocking NMIs. To inject an NMI into the rich OS/Apps, XMHF+'s NMI virtualization logic sets the NMI-window exiting bit in VMCS. This bit triggers an NMI window VM-exit when the rich OS/Apps unblock NMIs. In the NMI window VM-exit handler, XMHF+ injects the NMI into the rich OS/Apps.

One compatibility bug in XMHF arises from injecting NMIs into the rich OS-/Apps without checking whether NMIs are blocked by the rich OS/Apps. XMHF does not enable virtual NMIs, and its NMI virtualization logic injects NMIs into the rich OS/Apps directly, rather than using the NMI window VM-exit. Since Linux's NMI interrupt handler is not reentrant, injecting NMIs into Linux while NMIs are blocked can result in undefined behavior. For example, when we run Debian 11 as the rich OS in XMHF and frequently send NMIs to the CPU, we observe that Debian hangs in the NMI interrupt handler after XMHF injects the NMIs into it.

Simulating NMI Blocking

Another challenge is a race condition between the NMI interrupt handler and the non-NMI VM-exit handler that is interrupted by the NMI. Suppose the non-NMI VM-exit handler is modifying VMCS when an NMI arrives and the CPU invokes the

NMI interrupt handler. The NMI interrupt handler also modifies VMCS to inject the NMI into the rich OS/Apps. The race condition happens because both the non-NMI VM-exit handler and the NMI interrupt handler modify VMCS.

To prevent this race condition, the non-NMI VM-exit handler must identify critical sections in its logic. During the execution of any critical section, the NMI interrupt handler must not execute the NMI virtualization logic, which involves either executing the quiescing logic or modifying VMCS to inject the NMI into the rich OS/Apps, depending on whether quiescing is requested. XMHF+ simulates NMI blocking during critical section execution. Each CPU defines two per-CPU variables: B, a flag indicating whether the CPU is currently simulating NMI blocking, and V, a counter recording the number of NMIs that have visited the CPU but have not been handled by the NMI virtualization logic. XMHF+ initializes B to false and V to 0.

Algorithm 1 NMI Interrupt Handler

```
if B = true then
  V ⇐ V + 1
else {B = false}
  Execute NMI virtualization logic
end if
```

Algorithm 1 presents the implementation of the NMI interrupt handler. If the CPU is simulating NMI blocking (i.e., the critical section is running), the NMI interrupt handler increments V to indicate that an NMI has visited but has not been handled yet. The execution of the NMI virtualization logic is delayed until the critical section completes. Otherwise, the NMI interrupt handler executes the NMI virtualization logic. The NMI interrupt handler is not reentrant, and the hardware ensures that NMIs are blocked when the NMI interrupt handler is executing.

Algorithm 2 shows how the critical section is protected. Before executing the critical section, the non-NMI VM-exit handler sets B to true to simulate NMI blocking.

Algorithm 2 Protecting the NMI Critical Section

Require: B = false
Ensure: B = false
  $B \Leftarrow$ true
  Execute critical section
  $B \Leftarrow$ false
  while $V > 0$ **do**
    $V \Leftarrow V - 1$
    $B \Leftarrow$ true
    Execute NMI virtualization logic
    $B \Leftarrow$ false
  end while

Thus, if an NMI interrupt arrives, the NMI interrupt handler does not execute the NMI virtualization logic, ensuring that the critical section and the NMI virtualization logic are mutually exclusive. After the critical section completes, the algorithm checks whether V is greater than 0, indicating that one or more NMIs have arrived but have not been handled yet. If so, the algorithm calls the NMI virtualization logic to ensure that all NMIs are eventually handled.

In Algorithms 1 and 2, CPU-local variables B and V must be accessed atomically. Thus, we utilize x86's atomic instructions when increasing and decreasing V. We also use volatile variables and memory fences to prevent the compiler and the CPU from reordering or changing these algorithms.

This race condition results in a compatibility bug in XMHF. After the NMI interrupt handler modifies VMCS to inject the NMI interrupt into the rich OS/Apps, the non-NMI VM-exit handler that has been interrupted overwrites the modified VMCS fields. As a result, the NMI interrupt is never injected into the rich OS/Apps. Note that this bug does not violate memory integrity of the micro-hypervisor, as the NMI interrupt handler does not modify VMCS fields related to EPT.

3.3.5 Performance Improvement

We use multiple techniques to improve the performance of XMHF+. XMHF only supports compiling with GCC without optimization (O0). Without optimization, the code generated by GCC is inefficient. We fix a number of compile time and run time bugs to allow XMHF+ to compile with GCC's highest optimization (O3) and run correctly. This improves the performance of XMHF+.

XMHF+ also leverages several VMX features provided by the CPU to reduce the number of intercepts, which improves its performance since intercepts are expensive [19]. One of these features is the MSR bitmap, which XMHF+ uses to reduce the number of intercepts when the rich OS/Apps access MSRs. XMHF+ configures the MSR bitmap to allow the rich OS/Apps to access exported MSRs directly. In contrast, XMHF does not use the MSR bitmap, resulting in an intercept being generated every time the rich OS/Apps access an MSR. Therefore, XMHF+ generates fewer MSR intercepts than XMHF.

XMHF+ also enables VMCS shadowing, a VMX feature in modern CPUs. This feature allows XMHF+ to avoid unnecessary intercepts when the general-purpose hypervisor executes the read field from virtual-machine control structure (VMREAD) and write field to virtual-machine control structure (VMWRITE) instructions [20].

4

Evaluation

We evaluate XMHF+[1] through three dimensions. First, we measure the size of its code base. Second, we assess its compatibility by listing hardware and rich OS/Apps it supports. Third, we measure its performance through benchmarking.

When evaluating XMHF+'s performance, we first compile it in 32-bit and compare its performance with XMHF.[2] Then, we measure the performance of new features in XMHF+, including 64-bit support and virtualizing the hardware virtualization extension.

4.1 Code Size

We measure XMHF+'s TCB through its code size. We measure source lines of code (SLOC) using SLOCCount [21]. We report code size separately with and without

[1] During the entire evaluation, we use the XMHF+ source code as of February 15, 2023. The source code can be retrieved from `https://github.com/lxylxy123456/uberxmhf/tree/f6c71ded5c1541a6339614c5bad046c29671688d`.

[2] During the entire evaluation, we use XMHF version 6.1.0. The source code can be retrieved from `https://github.com/uberspark/uberxmhf/tree/3cd28bfe565221a6073950de4a8da9b61e10614e`.

	XMHF	XMHF+	XMHF+
VMX virtualization	No	No	Yes
top dir	25	61	61
xmhf-baseplatform	1229	1866	2076
xmhf-debug	123	213	213
xmhf-dmaprot	811	1742	1742
xmhf-eventhub	815	1735	1877
xmhf-memprot	538	951	954
xmhf-mm		884	884
xmhf-nested			4888
xmhf-partition	674	961	992
xmhf-smpguest	807	1089	1106
xmhf-startup	147	312	312
xmhf-tpm	445	135	135
xmhf-xcphandler	229	407	407
xmhf-xmhfcbackend	17	378	378
Total	5860	10734	16025

Table 4.1: The XMHF and XMHF+ core code sizes (in SLOCs).

VMX virtualization (i.e., virtualizing the hardware virtualization extension). Users who prefer a smaller TCB can configure XMHF+ during compile time to disable VMX virtualization.

The result is shown in Table 4.1. When not considering VMX virtualization, XMHF+'s SLOC increases by 4874 SLOCs (83%). Among the components that have the largest code size increase, the *dmaprot* component increases by 931 SLOCs because XMHF+ supports more IOMMUs and allows hypapps to use IOMMUs. The *eventhub* component increases by 920 SLOCs because XMHF+ is required to virtualize various hardware resources to achieve compatibility, as discussed in Section 3.3.2. The *mm* component contains a newly added memory allocator of 884 SLOCs. If needed, this component can easily switch to a more lightweight implementation to reduce code size.

VMX virtualization adds 5291 SLOCs to the XMHF+ core. VMCS virtualization

consists of around 2691 SLOCs. Intercept virtualization consists of around 1700 SLOCs. EPT virtualization consists of around 560 SLOCs.

We compare XMHF+'s VMX virtualization code size with KVM's nested virtualization in Linux 5.10.84. KVM does not clearly separate code for nested virtualization from code for single level of virtualization, so we only consider source file names that clearly indicate nested virtualization, like `nested.c` and `vmcs12.c`. We conservatively estimate that KVM's VMX nested virtualization code consists of at least 5095 SLOCs, which is similar to XMHF+'s VMX virtualization code size. KVM virtualizes EPT using a generic MMU component, which consists of an additional 6948 SLOCs.

4.2 Compatibility

We list the rich OSes that are compatible with XMHF+ and the hardware platforms on which XMHF+ can run. Since XMHF+ adds support for nested virtualization, we also list the general-purpose hypervisors that can run in XMHF+.

4.2.1 Rich OS Compatibility

[8] evaluates XMHF using Ubuntu 12.04 LTS, which is released in 2012. However, this version of Ubuntu is no longer supported by its vendor, which raises practicality and security challenges. As shown in Table 4.2, XMHF does not support modern operating systems such as Debian 11 and Windows 10. Many modern operating systems require 64-bit mode, but XMHF only supports 32-bit mode.

In contrast, XMHF+ supports a number of modern operating systems. For Linux distributions, XMHF+ supports Debian 11 and Fedora 35. We anticipate that XMHF+ supports other Linux distributions because all Linux distributions share the same kernel code. For Windows, XMHF+ supports recent Windows versions from Windows 7 to Windows 10. Unfortunately, XMHF+ does not support Windows 11,

Rich OS	Bit Size	Linux kernel	Release Year	Support End	XMHF	XMHF+
Ubuntu 12.04 LTS	32-bit	3.2.0-150	2012	2019	✓	✓
Debian 11	32-bit	5.10.0-21	2021	Supported		✓
Debian 11	64-bit	5.10.0-21	2021	Supported		✓
Fedora 35	64-bit	5.14.10-300	2021	2022		✓
Windows XP SP3	32-bit		2008	2014	✓	✓
Windows XP SP1	64-bit		2003	2014		✓
Windows 7	32-bit		2009	2020		✓
Windows 7	64-bit		2009	2020		✓
Windows 8.1	32-bit		2013	2023		✓
Windows 8.1	64-bit		2013	2023		✓
Windows 10	32-bit		2014	Supported		✓
Windows 10	64-bit		2014	Supported		✓
Windows 11	64-bit		2021	Supported		

Table 4.2: Rich OS compatibility of XMHF and XMHF+.

Model	CPU	CPU Frequency	Release Year	XMHF	XMHF+
HP EliteBook 2540p	i5-540M	2.53GHz	2010	✓	✓
Dell OptiPlex 7050	i5-7600	3.50GHz	2017		✓
HP EliteBook 840 Aero G8	i7-1185G7	3.00GHz	2021		

Table 4.3: Hardware compatibility of XMHF and XMHF+.

which requires UEFI booting.

4.2.2 Hardware Compatibility

For hardware compatibility, we test XMHF and XMHF+ on three machines. The test results are shown in Table 4.3. XMHF can only run on the HP EliteBook 2540p. The newer machines use TPM 2.0, which is not supported by XMHF.

XMHF+ supports TPM 2.0, so it can run on both the HP EliteBook 2540p and

General-Purpose Hypervisor	Rich OS	XMHF	XMHF+
KVM 5.10.0-21	Debian 11		✓
VirtualBox 6.1.42	Debian 11		✓
VMware Workstation 16.2.4	Debian 11		✓
VirtualBox 7.0.2	Windows 10		✓
VMware Workstation 16.2.4	Windows 10		✓
Hyper-V	Windows 10		✓

Table 4.4: General-purpose hypervisor compatibility of XMHF and XMHF+.

the Dell OptiPlex 7050. However, XMHF+ cannot run on the HP EliteBook 840 Aero G8 because this machine requires UEFI booting, a feature that XMHF+ does not currently support.

XMHF+ can run in KVM by disabling certain features not supported by KVM, like DRTM. This allows more efficient development of XMHF+ and hypapps. Booting XMHF+ in KVM is faster compared to physical hardware, and KVM provides debugging support using GDB. We expect XMHF+ to be able to run in other hypervisors like VMware and VirtualBox, but we have not tested these configurations yet.

4.2.3 General-Purpose Hypervisor Compatibility

XMHF does not support virtualization of the hardware virtualization extension, therefore it cannot run general-purpose hypervisors. However, XMHF+ does support virtualization of the hardware virtualization extension. As shown in Table 4.4, multiple popular general-purpose hypervisors are compatible with XMHF+. XMHF+ supports cross-platform general-purpose hypervisors such as VirtualBox and VMware Workstation, as well as platform-specific general-purpose hypervisors such as KVM and Hyper-V.

Windows 10 allows the user to enable virtualization-based security (VBS), which

leverages the hardware virtualization extension to achieve security properties in Windows. Hyper-V serves as the hypervisor in VBS's architecture [22]. After virtualizing the hardware virtualization extension, XMHF+ is one step closer to supporting VBS. However, VBS also requires UEFI booting and IOMMU virtualization [23], which are not yet supported by XMHF+. Thus, we are unable to enable VBS in Windows 10 running in XMHF+.

4.3 Performance Compared to XMHF

We compare the performance of XMHF+ with XMHF to see whether XMHF+ introduces overhead to the system. To measure the performance, we benchmark the rich OS/Apps and TrustVisor PALs. When applicable, we use the performance of bare metal as the baseline.

To ensure fairness, we compile both XMHF and XMHF+ in 32-bit mode. As discussed in Section 3.3.5, XMHF does not support optimization, but XMHF+ allows the user to enable compiler optimization with GCC's O3 optimization. Thus, we separately report the performance of XMHF+ with and without optimization.

We run all experiments on an HP EliteBook 2540p with a quad-core i5-540M CPU running at 2.53 GHz. This machine has 4 GiB of memory and an HDD. We use Ubuntu 12.04 LTS with kernel version 3.2.0-150 (32-bit) as the rich OS. This configuration is known to be supported by both XMHF and XMHF+. Swapping is disabled in the rich OS to accurately measure the performance of memory.

4.3.1 Rich OS/Apps Benchmarks

To determine XMHF+'s efficiency, we benchmark the rich OS/Apps in XMHF+. We use sysbench [24] to measure CPU and memory performance. We use IOzone [25] to measure file IO performance.

The benchmark results are shown in Figure 4.1. We can see that the performance

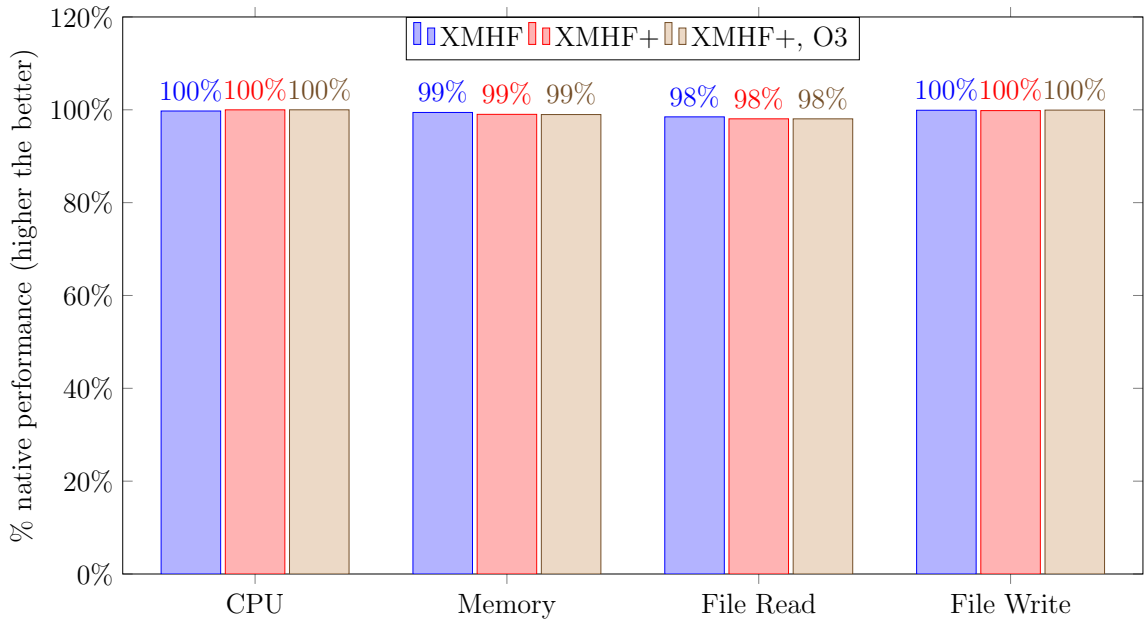

Figure 4.1: Rich OS/Apps performance on XMHF, XMHF+, and XMHF+ with O3 optimization. Normalized to native performance (running on bare metal).

of XMHF and XMHF+ is close. For all benchmarks, XMHF and XMHF+ achieve 98%–100% native performance.

4.3.2 TrustVisor Benchmarks

We design a custom microbenchmark to measure the efficiency of TrustVisor. The benchmark measures the average time TrustVisor takes to perform the four types of events during PALs' life cycle, as discussed in Section 2.2.2: registration, invocation, termination, and unregistration. We run the benchmark on XMHF and XMHF+ and compare the performance.

The benchmark results are shown in Table 4.5. The performance of XMHF and XMHF+ without optimization is close to each other. When compiled with O3 optimization, XMHF+ achieves a significant speedup of more than 150%.

Event	XMHF	XMHF+	XMHF+, O3
PAL Registration	964.6	919.0	271.5
PAL Invocation	105.0	118.9	46.0
PAL Termination	61.0	78.2	30.7
PAL Unregistration	112.4	117.8	37.8

Table 4.5: TrustVisor overhead on XMHF, XMHF+, and XMHF+ with O3 optimization. Measured in microsecond per event.

4.4 Performance of 64-bit and Virtualizing the Hardware Virtualization Extension

In this section, we measure the performance of XMHF+'s 64-bit support and hardware virtualization extension virtualization. We compare the performance of XMHF+ with other popular hypervisors: VirtualBox, VMware, and KVM. We compile XMHF+ using GCC's O3 optimization. We use a Dell OptiPlex 7050 with a quad-core i5-7600 CPU running at 3.50GHz and an HDD as the underlying hardware.

We use the same rich OS/Apps and TrustVisor benchmarks as in Section 4.3 to measure the performance. The rich OS used in all virtualization levels is Debian 11 with kernel version 5.10.0-21 (64-bit) and without a graphical user interface. Swapping is disabled to accurately measure the memory performance. The rich OS and guest VMs running the benchmarks always have 2 GiB of memory and 4 CPUs. For each level of virtualization, we add 1 GiB of memory to the rich OS to account for the memory overhead of the general-purpose hypervisor or micro-hypervisor. Table 4.6 shows the list of configurations we run.

4.4.1 Rich OS/Apps Benchmarks

We still use sysbench [24] to measure the CPU and memory performance and IOzone [25] to measure file IO performance.

First, we measure the performance of single level of virtualization. We measure

Name	L0/mL0	L1/mL1	L2
0	Debian (2 GiB)		
1b	Debian (3 GiB), VirtualBox	Debian (2 GiB)	
1k	Debian (3 GiB), KVM	Debian (2 GiB)	
1w	Debian (3 GiB), VMware	Debian (2 GiB)	
1x	XMHF+	Debian (2 GiB)	
2bk	Debian (4 GiB), VirtualBox	Debian (3 GiB), KVM	Debian (2 GiB)
2kk	Debian (4 GiB), KVM	Debian (3 GiB), KVM	Debian (2 GiB)
2wk	Debian (4 GiB), VMware	Debian (3 GiB), KVM	Debian (2 GiB)
2xk	XMHF+	Debian (3 GiB), KVM	Debian (2 GiB)

Table 4.6: Configurations for evaluating the performance of XMHF+.

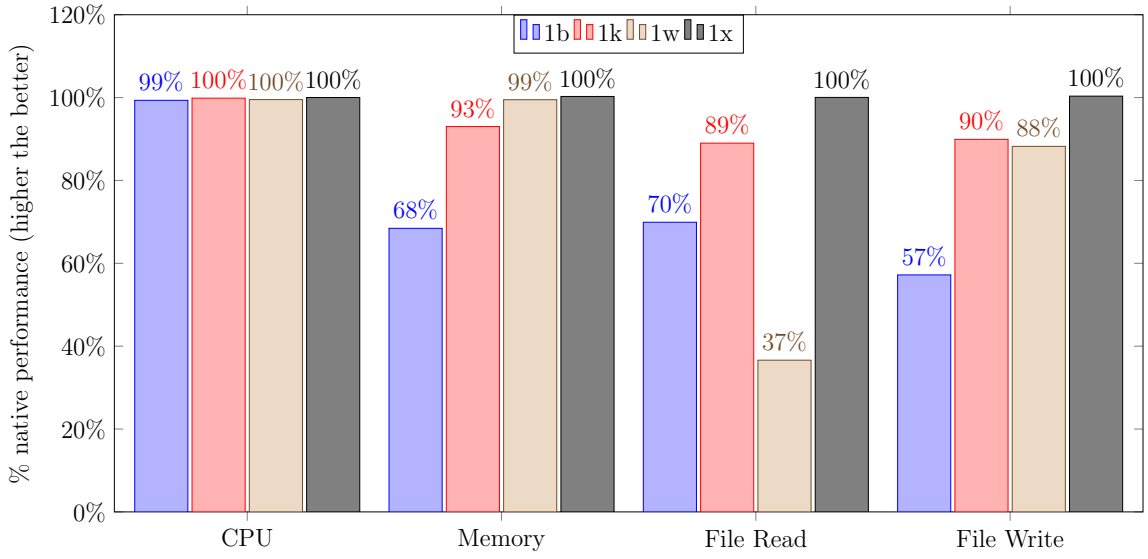

Figure 4.2: L1/mL1 Rich OS/Apps performance with single level of virtualization. XMHF+ is compiled in 64-bit mode with O3 optimization. Normalized to native performance (with configuration 0).

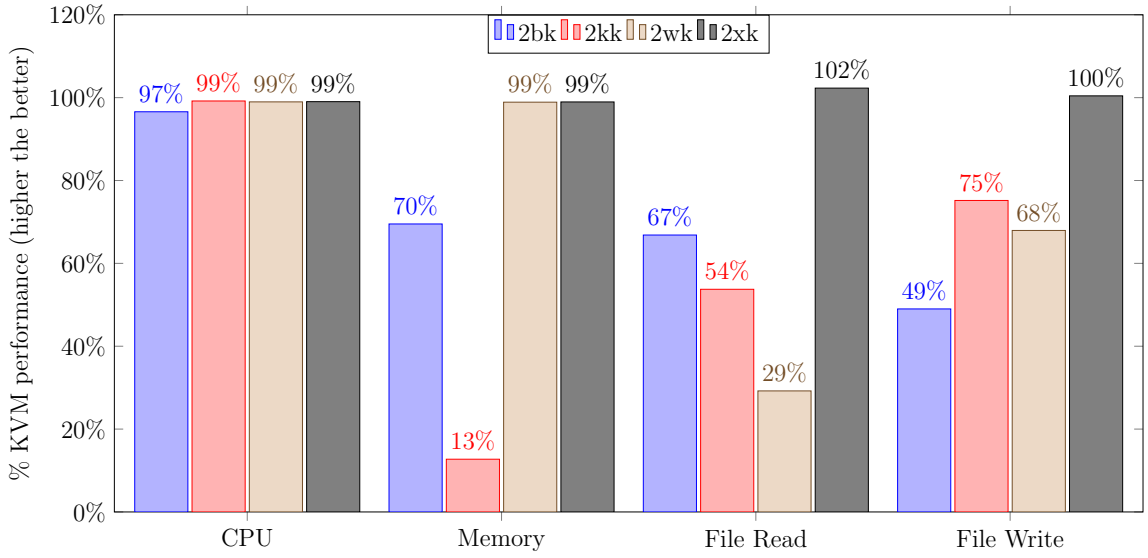

Figure 4.3: L2 Guest VMs performance with two levels of virtualization. KVM runs in L1/mL1. Different hypervisors or XMHF+ runs in L0/mL0. XMHF+ is compiled in 64-bit mode with O3 optimization. Normalized to KVM performance (with configuration 1k).

the performance of configurations 1b (VirtualBox), 1k (KVM), 1w (VMware), and 1x (XMHF+). We normalize all performance measurements to configuration 0 (running Debian on bare metal). The results are shown in Figure 4.2, which indicates that XMHF+ has close to 100% performance. The general-purpose hypervisors have notable performance overhead on memory and file IO benchmarks.

We then measure the performance of two levels of virtualization using KVM as the hypervisor in L1/mL1. We measure the performance of configurations 2bk (running VirtualBox in L0), 2kk (running KVM in L0), 2wk (running VMware in L0), and 2xk (running XMHF+ in mL0). To highlight the performance overhead of nested virtualization (for general-purpose hypervisors) and virtualizing the hardware virtualization extension (for XMHF+), we normalize all performance measurements to configuration 1k (running KVM on bare metal). The results are shown in Figure 4.3.

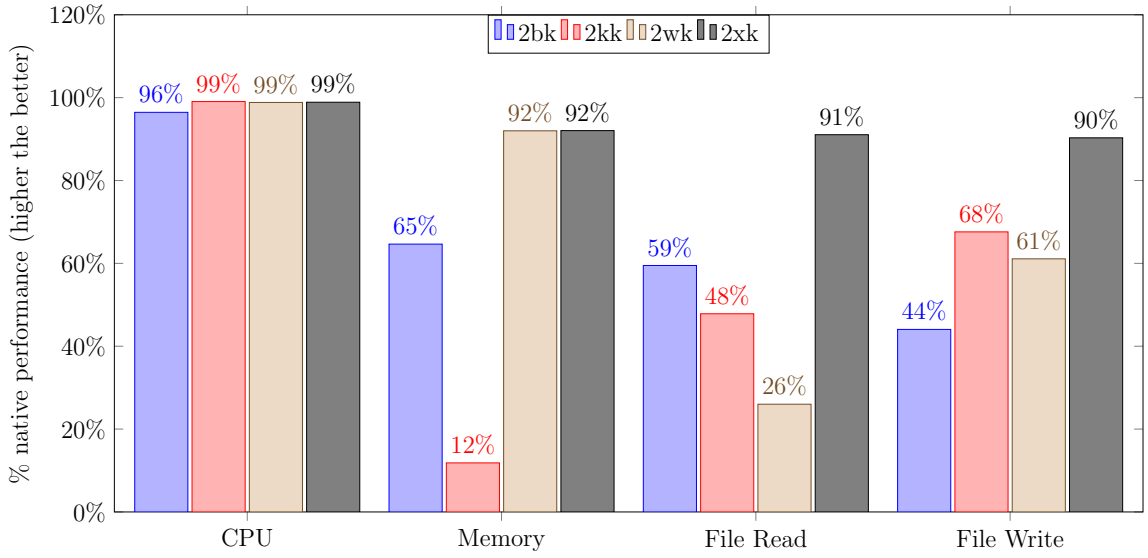

Figure 4.4: L2 Guest VMs performance with two levels of virtualization. KVM runs in L1/mL1. Different hypervisors or XMHF+ runs in L0/mL0. XMHF+ is compiled in 64-bit mode with O3 optimization. Normalized to native performance (with configuration 0).

We can see that XMHF+ still achieves close to 100% performance. However, for unknown reasons, our file IO benchmark reports slightly higher than 100% performance for 2xk compared to 1k. We leave the investigation of this phenomenon as future work. The general-purpose hypervisors have notable performance overhead on memory and file IO benchmarks.

In Figure 4.4, we compare the performance of configurations 2bk, 2kk, 2wk, and 2xk with native performance (i.e. configuration 0). We can see that XMHF+'s L2 guest VMs have 10% or less performance overhead compared to native performance. Note that this figure only shows the combined performance overhead of two levels of virtualization. The performance overhead of KVM, which runs in L1/mL1, is shown in Figure 4.2. The performance overhead of hypervisors and XMHF+, which run in L0/mL0, is shown in Figure 4.3.

Event	1x	2xk
PAL Registration	91.7	148.1
PAL Invocation	23.1	37.4
PAL Termination	15.8	21.4
PAL Unregistration	20.0	42.7

Table 4.7: TrustVisor overhead on XMHF+ (configuration 1x) and KVM in XMHF+ (configuration 2xk). XMHF+ is compiled in 64-bit mode with O3 optimization. Measured in microsecond per event.

4.4.2 TrustVisor Benchmarks

We run the same TrustVisor benchmark as in Section 4.3.2. We run the benchmark in configurations 1x (XMHF+ only) and 2xk (KVM in XMHF+). This allows us to compare the performance of interacting with PALs from guest VMs and from the rich OS/Apps.

The benchmark results are shown in Table 4.7. We can see that compared to configuration 1x, configuration 2xk takes approximately 110% more time to perform PAL unregistration and 50% more time to perform other events. Compared to the results in Section 4.3.2, TrustVisor has lower overhead because this benchmark runs on newer hardware.

5

TrustVisor Vulnerabilities

As discussed in Section 2.2.2, TrustVisor is a micro-hypervisor ported to run on XMHF [8]. XMHF version 6.1.0 provides TrustVisor as an example hypapp. During the development of XMHF+, we find six security vulnerabilities in TrustVisor when running as a hypapp in XMHF.[1] These vulnerabilities are fixed in XMHF+, and we backport these fixes to XMHF.

Note that the following vulnerabilities are only what we find while working on this thesis. We do not claim that XMHF, XMHF+, and TrustVisor have no other vulnerabilities.

5.1 TrustVisor Vulnerability 1: INIT Interrupt During Restart

In the x86 micro-architecture, when VMX is disabled, the INIT interrupt resets the CPU. It is commonly used to initialize symmetric multiprocessing (SMP), as depicted in Figure 5.1. The procedure comprises of two steps. First, CPU 1 sends an INIT interrupt to CPU 2 to reset it. Second, CPU 1 sends a start-up interprocessor

[1] The vulnerabilities we find may or may not apply before TrustVisor is ported to run in XMHF, i.e., when TrustVisor runs as a standalone micro-hypervisor.

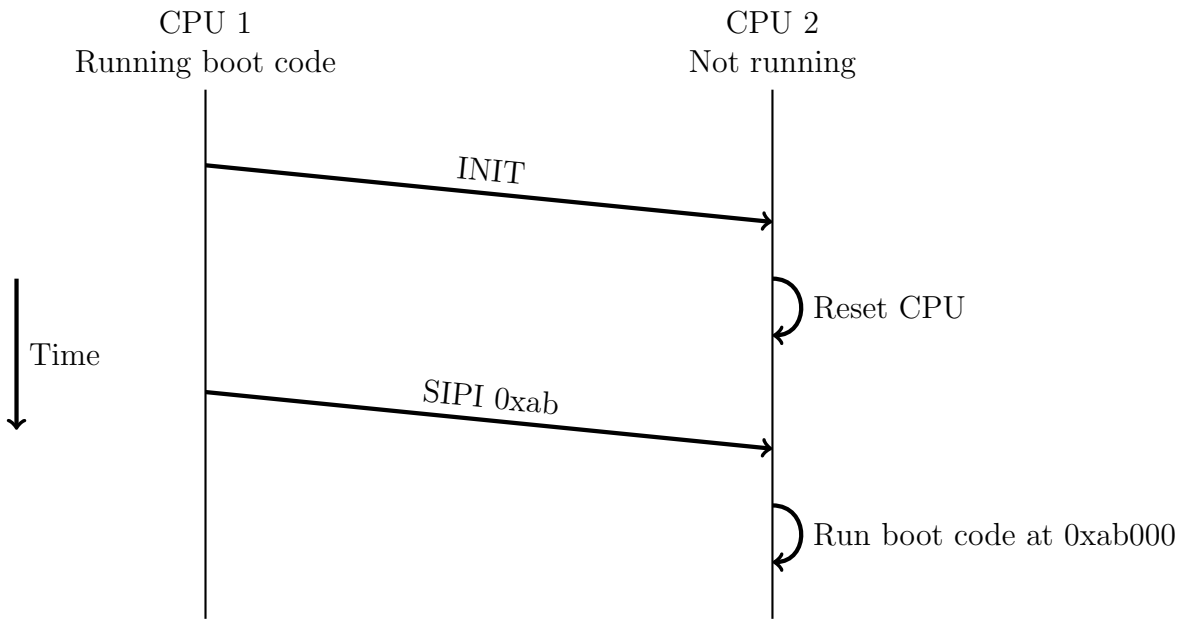

Figure 5.1: Use of INIT and SIPI interrupts during SMP boot.

interrupt (SIPI) to CPU 2 to boot it. The SIPI interrupt includes the location of the boot code to be executed on CPU 2 [9].

The behavior of INIT interrupts changes when VMX is enabled, as defined in [9]. When the CPU is running in host mode (i.e., XMHF and TrustVisor), INIT interrupts are blocked by hardware. When the CPU is running in guest mode (i.e., rich OS/Apps and isolated code), INIT interrupts cause intercepts.

The restart process of XMHF and TrustVisor is illustrated in Figure 5.2. First, the rich OS sends a restart request to one of the available hardware, such as the PS/2 keyboard controller or the ACPI reset register [17]. Although the rich OS can send the restart request to any hardware, XMHF assumes that the hardware tries to reset all CPUs by sending them INIT interrupts. Since VMX is enabled on all CPUs, the INIT interrupts trigger INIT intercepts in XMHF. On each CPU, XMHF's INIT intercept handler invokes TrustVisor's restart callback function. This function

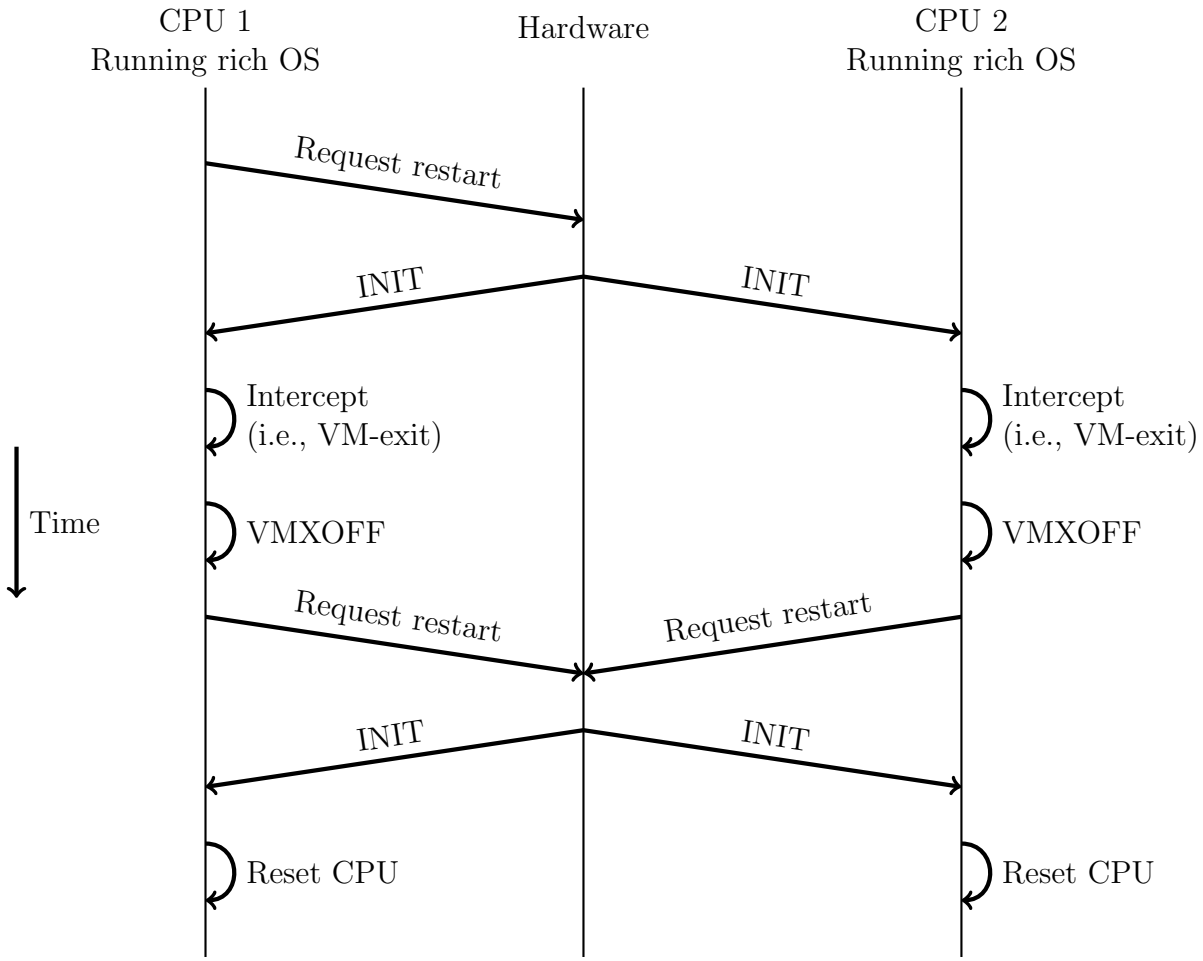

Figure 5.2: Normal XMHF and TrustVisor restart process.

disables VMX on the CPU using the leave VMX operation (VMXOFF) instruction and sends a restart request to the PS/2 keyboard controller. Since VMX is disabled, the hardware can successfully reset all CPUs through INIT interrupts.

As illustrated in Figure 5.3, the security vulnerability arises when CPU 2 disables VMX but has not requested the PS/2 keyboard controller to restart, and CPU 1 sends an INIT interrupt to CPU 2. This results in resetting CPU 2. CPU 1 can then send an SIPI interrupt to boot the CPU 2 with malicious code, which can read and write any memory, including the micro-hypervisor's memory and PALs' memory. This is

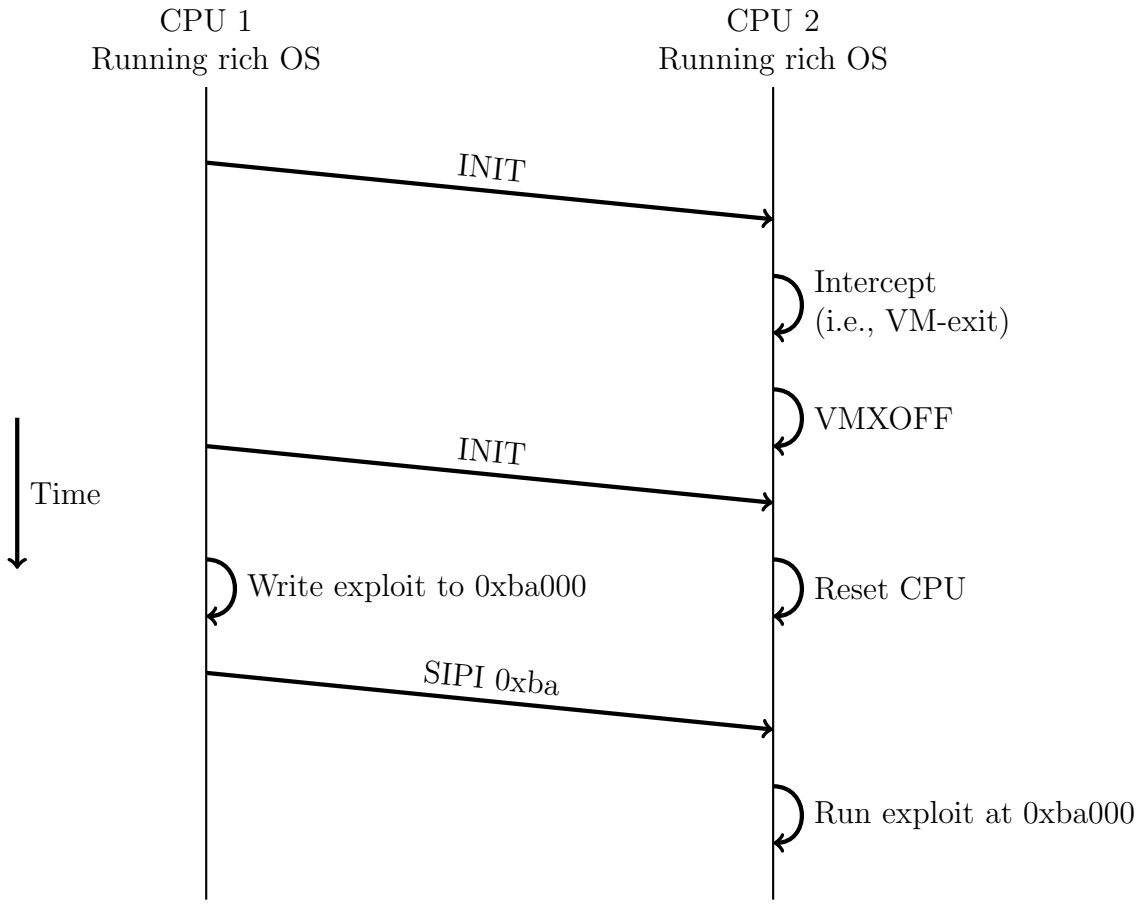

Figure 5.3: Exploiting the TrustVisor vulnerability using INIT interrupts.

not a vulnerability of XMHF because TrustVisor's restart callback function fails to quiesce other CPUs.

This vulnerability can only be exploited when DRTM is disabled. DRTM in Intel CPUs is implemented through Intel Trusted Execution Technology (TXT). In DRTM, when one CPU with VMX disabled receives an INIT interrupt, Intel TXT shutdown is triggered and all CPUs are reset [26].

To fix this vulnerability, TrustVisor synchronizes all CPUs before disabling VMX. This way, the attacker cannot send the SIPI interrupt to any CPU after VMX is disabled, thereby preventing the execution of attacker-controlled code in host mode.

5.2 TrustVisor Vulnerability 2: Intel TXT Secret Setting

In DRTM, software can set the TXT.CMD.SECRETS flag to indicate that memory contains secret value. When this flag is set and Intel TXT shutdown happens, the hardware clears the memory to protect secrets [26].

TrustVisor must ensure the memory secrecy of PALs. However, we find that when DRTM is enabled, TrustVisor fails to set the TXT.CMD.SECRETS flag. To exploit this vulnerability, an attacker can register a PAL and then restart the machine without unregistering the PAL. As the memory does not change after the machine restarts, the attacker can boot a malicious OS and dump the memory, which contains the secret memory of the PAL.

To fix this vulnerability, we set the TXT.CMD.SECRETS flag during TrustVisor's initialization.

5.3 TrustVisor Vulnerability 3: PALs not Destroyed When Restarting

To protect the memory secrecy of PALs, TrustVisor clears their memory when they are unregistered. However, we find that if the rich OS restarts while some PALs are still registered, TrustVisor does not clear the PALs' memory in its restart callback function. Similar to the exploit of TrustVisor vulnerability 2 in Section 5.2, an attacker can boot a malicious OS and dump the secret memory of PALs.

Note that this vulnerability is distinct from TrustVisor vulnerability 2. The latter applies only when DRTM is enabled and assumes that the attacker can bypass the restart callback function when restarting the machine, for example, by pressing the restart button on the hardware. On the other hand, this vulnerability applies when DRTM is disabled and assumes that the attacker cannot bypass the restart callback function in TrustVisor. Nevertheless, DRTM is essential to defend against an attacker who can bypass the restart callback function.

To fix this vulnerability, we add the logic to clear the memory of all registered PALs in TrustVisor's restart callback function.

5.4 TrustVisor Vulnerability 4: Incorrect Memory Access Check during PAL Registration

During PAL registration, the rich OS/Apps specify the memory regions of the PAL. TrustVisor must check that the rich OS/Apps are allowed to access the memory regions before adding them to the PAL. However, we discover a typo in this check that allows the rich OS/Apps to specify memory regions that are inaccessible to them. As a result, an attacker can register a malicious PAL and read and write memory of other PALs, TrustVisor, and XMHF from within the malicious PAL. This violates the memory separation property of TrustVisor and the memory integrity of the micro-hypervisor.

To fix this vulnerability, we fix the typo in the check during PAL registration.

5.5 TrustVisor Vulnerability 5: Race Condition when Switching EPTP

By default, XMHF creates separate EPTs for each CPU. However, for implementation convenience, TrustVisor uses the same EPT for all CPUs. When the first PAL is registered, TrustVisor quiesces all other CPUs and modifies their EPT pointers (EPTPs) to point to the same EPT. However, if a CPU is in the process of modifying its EPTP when being quiesced, TrustVisor may fail to modify the CPU's EPTP due to race condition. Consequently, the rich OS can read and write the memory of PALs from the CPU whose EPTP is not modified by TrustVisor.

To fix this vulnerability, we avoid updating EPTPs of one CPU from another CPU. As a workaround, we update the EPT page map level 4 entries (PML4Es) instead. However, we argue that in the long term, it is better for TrustVisor to allow different EPTs for each CPU.

5.6 TrustVisor Vulnerability 6: Side Channel in Registers

We find that during PAL termination, TrustVisor does not clear the CPU registers that may contain secrets of the PAL. This creates a side channel that allows a malicious rich OS/Apps to violate the secrecy of the PAL. As discussed in Section 2.2.2, the rich OS/Apps call the PAL through a C function call. According to the 32-bit x86 calling convention [27], registers can be categorized into three groups.

- **Callee-saved registers**: For 32-bit PALs, callee-saved registers include EBX, ESI, EDI, and EBP [27]. During PAL termination, the calling convention requires all callee-saved registers to contain the same value as the value during PAL invocation. TrustVisor assumes that PALs are compiled by a benign compiler following the calling convention, so callee-saved registers do not leak any secret information.

- **Scratch register for return values**: For 32-bit PALs, the scratch register for return value is EAX [27].[2] During PAL termination, this register contains the return value of the PAL. Thus, the scratch register for return values does not leak any secret information.

- **Scratch registers not for return values**: For 32-bit PALs, scratch registers not for return values include ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, and K0-K7 [27]. During PAL termination, the calling convention allows these registers to contain any value. For example, a benign compiler may compile a PAL that uses these registers to compute secret information in the PAL. However, the PAL does not clear these registers when it returns. Thus, a malicious rich OS/Apps can learn about the secret

[2] We assume that PALs only return integers, so the ST(0), XMM0, YMM0, and ZMM0 registers are not for return values.

by reading these registers after PAL termination.

To fix this vulnerability, we clear all general-purpose registers (GPRs) that are scratch registers not for return values when the PAL terminates. For 32-bit PALs, we clear ECX and EDX. For 64-bit PALs, we clear RCX, RDX, RSI, RDI, and R8-R11 [27]. When the PAL is executing, we disable access to the FPU and SIMD instructions. Thus, the PAL cannot access other scratch registers not for return values (i.e., ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7, and K0-K7).

6

Discussions and Future Work

In this section, we discuss miscellaneous findings during this research and future work.

6.1 Limitation in Quiescing

As discussed in Section 2.3, the formal verification of XMHF's and its hypapp's memory integrity requires all intercept handlers to run in a single-threaded environment. The XMHF design achieves atomicity of intercept handlers through quiescing: when any CPU runs an intercept handler, all other CPUs are quiesced.

However, we find an implementation limitation in XMHF version 6.1.0. Specifically, XMHF does not achieve atomicity by quiescing all other CPUs during intercept handlers. Instead, it uses two strategies to prevent race conditions between CPUs. First, when the intercept handler calls the hypapp callback function, XMHF quiesces all other CPUs. This prevents race conditions when the hypapp callback function concurrently accesses shared states on multiple CPUs. Second, XMHF uses a spin lock to protect the printf function from concurrent access. This prevents race conditions when multiple CPUs print debug messages at the same time. Since other states

and resources accessed by the intercept handler are specific to the current CPU, race conditions do not occur in these cases.

In XMHF+, we temporarily inherit this implementation limitation from XMHF. The XMHF+ intercept handler quiesces all other CPUs before calling hypapp callback functions. XMHF+ uses spin locks and read-write locks to prevent race conditions when the intercept handler accesses states and resources shared between CPUs. Other states and resources accessed by the intercept handler are specific to the current CPU, where race conditions do not occur.

We argue that this is only a limitation in the current implementation of XMHF+. All XMHF+ intercept handlers can run without interaction with other CPUs, so it is relatively easy to modify the XMHF+ implementation to quiesce all other CPUs when any intercept handler is running. On modern hardware, we only expect a minor decrease in performance. We leave this modification as future work.

6.2 Incompatibility due to Hardware and Hypapp

6.2.1 Incompatibility due to Hardware

XMHF and XMHF+ assume that the hardware is backward-compatible. For example, the x86 instruction set architecture (ISA) is backward-compatible. Thus, if XMHF and XMHF+ can run on an old x86 CPU, ideally they should be able to run on a new x86 CPU without modification.

Unfortunately, modern hardware makes two changes that break backward compatibility, affecting the compatibility of XMHF and XMHF+. First, modern hardware updates from TPM 1.2 to TPM 2.0, which is not backward-compatible. However, XMHF only supports TPM 1.2, so it cannot run on modern hardware that uses TPM 2.0. In XMHF+, we update the TPM-related code to support TPM 2.0.

Second, modern hardware introduces backward compatibility issues in the boot process. There are two methods for booting an operating system on x86 hardware:

Compatibility Support Module (CSM) booting and UEFI booting. While older hardware supports both CSM and UEFI booting, newer hardware only supports UEFI booting [28]. Currently, XMHF and XMHF+ only support CSM booting, which makes them incompatible with the latest hardware. We plan to add UEFI booting support in XMHF+ in future work.

6.2.2 Incompatibility due to Hypapp

We find another compatibility issue due to TrustVisor. TrustVisor assumes that once a PAL is created, the rich OS/Apps do not access the memory of the PAL. However, sometimes the SysMain service in Windows 10 accesses the PAL's memory, which violates TrustVisor's memory separation property. Windows 10 uses this service to perform a novel physical memory management technique called SuperFetch [22]. In the short term, we fix this problem by disabling the SysMain service in Windows 10 before running PALs. However, we argue that in the long term, TrustVisor should relax its assumptions to accommodate modern OSes.

6.3 Bugs in Other Software

We find multiple bugs in other software, such as compilers and general-purpose hypervisors. Additionally, we discover a few typos in hardware manuals. However, most of these bugs cause compatibility issues that only arise in uncommon scenarios and, to our knowledge, do not have security implications. For example, we find that KVM cannot run some L2 guests we develop correctly. Nevertheless, this bug does not affect widely used guest operating systems such as Windows and Linux.

Appendix A contains a list of the bugs we report, and some of them have already been fixed by the maintainers.

6.4 Future Work

6.4.1 Compatibility

In the future, we plan to support IOMMU virtualization and UEFI booting in XMHF+. Currently, XMHF+ hides the IOMMU from the rich OS/Apps because general-purpose hypervisors do not require it. However, virtualization-based security (VBS) in Windows 10 requires the IOMMU [23]. UEFI booting is also required by VBS and the latest operating systems like Windows 11. After supporting IOMMU virtualization and UEFI booting, we will work on enabling VBS in Windows 10.

Though XMHF supports both Intel and AMD CPUs, XMHF+ currently only supports Intel CPUs. We plan to add support to XMHF+ for AMD CPUs in the future. The hardware virtualization extension in AMD CPUs is provided through Secure Virtual Machine (SVM), a technology different from VMX. Thus, XMHF+ must also virtualize SVM to support AMD CPUs.

6.4.2 Formal Verification

The formal verification of XMHF+'s and its hypapp's memory integrity is left as future work. However, as shown in Section 3.2.3, XMHF+ satisfies all properties required by the DRIVE methodology. [8] formally proves the memory integrity of XMHF and its hypapp using CBMC, and we believe that a similar approach can be followed to prove the memory integrity of XMHF+ and its hypapp.

While formally proving the memory integrity of XMHF+ and its hypapp is important, memory integrity alone cannot guarantee the security of XMHF+. Similar to XMHF, XMHF+ also needs to maintain its control flow integrity (CFI) to ensure its security [8]. Therefore, we expect future work to address other security properties of XMHF+, including CFI.

6.4.3 Hypapps Support

Currently, XMHF+ only supports TrustVisor, but supporting other hypapps is left as future work. To enable a general hypapp to interact correctly with guest VMs, it may be necessary to modify the interface between XMHF+ and hypapps. For example, XMHF+ may need to define new callback functions.

We also find that some hypapps may not function correctly under XMHF and XMHF+'s quiescing design. For example, if a hypapp's callback function requires interaction with another CPU, the hypapp cannot run correctly because all other CPUs are quiesced. In the future, we plan to investigate how XMHF+ can support this type of hypapp while maintaining its atomicity.

6.4.4 Performance Optimization

In the future, we plan to further increase the performance of XMHF+ by optimizing the use of EPTs. Currently, XMHF+ manages all rich OS/Apps memory with 4 KiB pages in EPT, resulting in significant memory overhead. For a machine with 8 GiB of memory and eight CPUs, XMHF+ requires more than 128 MiB to store EPT. However, if XMHF+ uses 2 MiB or 1 GiB pages in EPT, the memory overhead would be less than 0.5 MiB. Using large pages also improves performance since EPT uses fewer TLB entries.

6.5 Development Process

We are able to speed up the development of XMHF+ by running it in general-purpose hypervisors like KVM [10, 29]. KVM can simulate many hardware resources used by XMHF+, including the hardware virtualization extension and the IOMMU. Virtual machines in KVM can be debugged using GDB, allowing us to view XMHF+'s registers and control its execution. In contrast, when developing on bare metal, we have to rely on print debugging. Debugging with KVM and GDB is especially helpful

for us when developing the boot process and resolving race conditions. KVM also boots virtual machines faster than real hardware, which saves us time when testing bug fixes for XMHF+.

We use continuous integration (CI) to automatically catch regression problems. We set up CI pipelines to run XMHF+ and TrustVisor in KVM and test their basic functionalities. We use Jenkins [30] as the platform to run CI pipelines on our local machine and CircleCI [31] to run CI pipelines on the cloud. CircleCI supports nested virtualization, allowing us to run KVM on CircleCI. However, due to the overhead of nested virtualization in CircleCI, we need to compile XMHF+ with the highest optimization.

Open source software helps us when developing XMHF+. When XMHF+ runs open source rich OS/Apps and general-purpose hypervisors like Linux and KVM, the entire software stack is transparent to us. For example, if Linux crashes as a rich OS in XMHF+, we can debug Linux to see how XMHF+ triggers the incorrect behavior in Linux. In return, we help the open source community by filing detailed bug reports. For example, for a few KVM bugs we encounter, we debug KVM and include the root causes of the bugs in the bug reports.

7

Conclusion

Micro-hypervisors are used in many research projects to improve the security of computer systems. For example, micro-hypervisors can separate security-sensitive programs from a commodity operating system, which typically contains millions of lines of code. XMHF [8] is a framework that facilitates the development of micro-hypervisors. However, the hardware and operating systems XMHF supports are obsolete.

In this thesis, we demonstrate that micro-hypervisors can support modern hardware and operating systems. We present XMHF+, an enhanced version of XMHF. XMHF+ satisfies the micro-architectural requirements of modern rich OS/Apps (e.g., Windows 10, Debian 11) by authorizing their accesses to hardware resources with three policies: exporting, hiding, and virtualization. XMHF+ also virtualizes the hardware virtualization extension to support rich OS/Apps to run general-purpose hypervisors. Moreover, XMHF+ supports 64-bit micro-architectures and TPM 2.0.

Our compatibility improvements to XMHF+ do not compromise its design for verifiability. XMHF+ preserves all security properties required by the DRIVE methodology, which is used to prove the memory integrity of XMHF and its hy-

papp [8]. Thus, the memory integrity of XMHF+ and its hypapp can be formally verified in future work.

In this thesis, we identify six vulnerabilities in TrustVisor when running as a hypapp in XMHF. We address these vulnerabilities in XMHF+ and backport the fixes to XMHF. Additionally, we report multiple bugs in other software used during our development, such as compilers and general-purpose hypervisors.

Bibliography

[1] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.

[2] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a safe and practical architecture for security applications on commodity platforms," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 34–54.

[3] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 209–222.

[4] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 308–323.

[5] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 616–630.

[6] M. Yu, V. D. Gligor, and Z. Zhou, "Trusted display on untrusted commodity platforms," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 989–1003.

[7] J. Cui, Y. Zhang, Z. Cai, A. Liu, and Y. Li, "Securing display path for security-sensitive applications on mobile devices," *Computers, Materials and Continua*, vol. 55, no. 1, p. 17, 2018.

[8] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 430–444.

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., Santa Clara, CA, 2021.

[10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[11] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[12] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 87–104.

[13] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the twenty-third acm symposium on operating systems principles*, 2011, pp. 203–216.

[14] Assured Information Security, "Bareflank," 2021, Accessed: Apr. 4, 2023. [Online]. Available: https://github.com/Bareflank/hypervisor.

[15] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[17] Unified Extensible Firmware Interface Forum, "Acpi specification 6.5," 2022, Accessed: Apr. 4, 2023. [Online]. Available: https://uefi.org/specs/ACPI/6.5/.

[18] D. P. Bovet and M. Cesati, "Checking the nmi watchdogs," in *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005, sec. 6.4.4.

[19] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 373–385.

[20] Intel, "4th generation intel core vpro processors with intel vmcs shadowing," 2013, Accessed: Feb. 14, 2023. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf.

[21] D. A. Wheeler, "SLOCCount (2.26)," 2001, Accessed: Apr. 4, 2023. [Online]. Available: https://dwheeler.com/sloccount/.

[22] P. Yosifovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Microsoft Press, 2017.

[23] Microsoft, "Virtualization-based security (vbs)," Jun. 2022, Accessed: Jan. 18, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[24] A. Kopytov *et al.*, "sysbench (1.0.20)," 2020, Accessed: Apr. 4, 2023. [Online]. Available: https://github.com/akopytov/sysbench.

[25] W. D. Norcott, "Iozone filesystem benchmark," 2003, Accessed: Apr. 4, 2023. [Online]. Available: http://www.iozone.org/.

[26] *Intel Trusted Execution Technology (Intel TXT) Software Development Guide*, Intel Corp., Santa Clara, CA, 2022.

[27] A. Fog, "Calling conventions for different c++ compilers and operating systems," 2022, Accessed: Apr. 4, 2023. [Online]. Available: https://www.agner.org/optimize/calling_conventions.pdf.

[28] Intel, "Legacy bios boot support removal for intel platforms," Jul. 2020, Accessed: Jan. 16, 2023. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/intel-nuc/Legacy-BIOS-Boot-Support-Removal-for-Intel-Platforms.pdf.

[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Ottawa, Ontario, Canada, 2007, pp. 225–230.

[30] Jenkins, "Jenkins," 2023, Accessed: Apr. 4, 2023. [Online]. Available: https://www.jenkins.io/.

[31] CircleCI, "CircleCI," 2023, Accessed: Apr. 4, 2023. [Online]. Available: https://circleci.com/.

Appendix A

Reported Bugs

As mentioned in Section 6.3, we report multiple bugs in software and hardware manuals. Table A.1 contains a list of bugs we find. Note that not all bugs are confirmed by the maintainers. Some bugs are considered limitations of the software, and we may also incorrectly report false positives.

Vendor	Bug Description
QEMU[1]	Assertion error related to break points while debugging KVM[2]
QEMU	Assertion error while debugging SMM code in KVM[3]
KVM[4]	Incorrect check in nested virtualization[5]
KVM	EOI may be ignored in x2APIC[6]
KVM	KVM bug detected in nested virtualization[7]
KVM	Some VMCS fields are missing from VMCS12[8]
KVM	Incorrect handling of PDPTE in nested virtualization[9]
KVM	Incorrect handling of REP INS instructions in nested virtualization[10]
lmm[11]	Assertion error in memory allocator[12]
Pathos[13]	Incorrect handling of spurious IRQ[14]
GCC[15]	False positive in static analysis[16]
GCC	False negative in static analysis[17]
KVM	Incorrect handling of NMI blocking in nested virtualization[18]
Bochs[19]	Incorrect handling of NMI blocking in VMX emulation[20]
VMware[21]	Incorrect handling of NMI blocking in nested virtualization[22]
KVM	Incorrect emulation of LOCK instruction in MMIO[23]
VMware	Incorrect emulation of LOCK instruction in MMIO[24]
Intel[25]	Typo in Software Developer Manuals (SDM)[26]
Intel	Typo in Intel TXT software development guide[27]

Table A.1: List of software and hardware manual bugs reported.

[1] QEMU is the frontend for KVM. See `https://www.qemu.org/`.

[2] Report is available at `https://gitlab.com/qemu-project/qemu/-/issues/1045`.

[3] Report is available at `https://gitlab.com/qemu-project/qemu/-/issues/1047`.

[4] KVM is a hypervisor. See `https://www.linux-kvm.org/page/Main_Page`.

[5] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216033`.

[6] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216045`.

[7] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216046`.

[8] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216091`.

[9] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216212`.

[10] Report is available at `https://bugzilla.kernel.org/show_bug.cgi?id=216234`.

[11] The List Memory Manager (lmm) provides a simple memory management service. See https://github.com/OSPreservProject/oskit.

[12] Report is available at https://github.com/OSPreservProject/oskit/issues/1.

[13] Pathos is the reference kernel implementation used in 15-410 (Operating System Design and Implementation) at Carnegie Mellon University (CMU). See https://www.cs.cmu.edu/~410/.

[14] Reported through private channel.

[15] GCC is a compiler. See https://gcc.gnu.org/.

[16] Report is available at https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105100.

[17] Report is available at https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107663.

[18] Report is available at https://bugzilla.kernel.org/show_bug.cgi?id=217304.

[19] Bochs is an x86 simulator. See https://bochs.sourceforge.io/.

[20] Report is available at https://sourceforge.net/p/bochs/bugs/1456/.

[21] VMware is a hypervisor. See https://www.vmware.com/content/vmware/vmware-published-sites/us/products/workstation-pro.html.

[22] Report is available at https://communities.vmware.com/t5/VMware-Workstation-Pro/Bug-NMI-incorrectly-blocked-in-guest-if-host-blocks-NMI-and/m-p/2964057.

[23] Report is available at https://bugzilla.kernel.org/show_bug.cgi?id=216867.

[24] Report is available at https://communities.vmware.com/t5/VMware-Workstation-Player/LOCK-instruction-atomicity-broken-on-VGA-memory-mapped-IO/m-p/2946505.

[25] Intel is an x86 CPU manufacturer. See https://www.intel.com/content/www/us/en/resources-documentation/developer.html.

[26] Report is available at https://community.intel.com/t5/Processors/Typo-in-SDM-volume-4/m-p/1391506.

[27] Report is available at https://community.intel.com/t5/Intel-Trusted-Execution/Typos-in-TXT-development-guide-315168-017/m-p/1422314.