"*It's an app. It's a hypervisor. It's a hypapp.*": Design and Implementation of an eXtensible and Modular Hypervisor Framework

Amit Vasudevan, Jonathan M. McCune, and James Newsome

June 26, 2012

CMU-CyLab-12-014

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

"It's an app. It's a hypervisor. It's a hypapp.": Design and Implementation of an eXtensible and Modular Hypervisor Framework*

Amit Vasudevan
CyLab, Carnegie Mellon University
amitvasudevan@acm.org

Jonathan M. McCune
CyLab, Carnegie Mellon University
jonmccune@cmu.edu

James Newsome
CyLab, Carnegie Mellon University
jnewsome@cmu.edu

ABSTRACT

This paper presents our efforts in developing XMHF, an eXtensible and Modular Hypervisor Framework. XMHF takes a *developer-centric* approach to hypervisor design and implementation, and strives to be a comprehensible and flexible platform for performing hypervisor research and development. XMHF encapsulates common hypervisor core functionality in a framework that allows others to build custom hypervisor-based solutions (called "hypapps") while freeing them from a considerable amount of wheel-reinventing that is often associated with such efforts. We are encouraged by the end result – a clean, barebones hypervisor framework with desirable performance characteristics and an architecture amenable to formal analysis.

Keywords

eXtensible Modular Hypervisor Framework, hypervisor-based applications, hypapps, dynamic root of trust, nested (2-dimensional) paging
*

1. INTRODUCTION

Recent years have yielded significant research on hypervisor-based architectures for security [9,15,17,24–26,29,30,32,34, 37,38,40,41,45,46]. A majority of these hypervisors [17,24, 25,29,37,38,40,41,46] are designed and written from scratch with the primary goal of achieving a low Trusted Computing Base (TCB) while providing a specific security property. Other research efforts leverage existing commercial-grade virtualization solutions (e.g., Xen, Linux KVM, VMware, or L4), but generally do not require such rich functionality [9,12,15,19,26,30,32,34,45].

This paper presents our efforts in developing XMHF, an eXtensible and Modular Hypervisor Framework. XMHF takes a *developer-centric* approach to hypervisor design and implementation, and strives to be a comprehensible and flexible platform for performing hypervisor research and development. We are motivated in part by the fact that every hypervisor-based solution relies on a common hypervisor core functionality that is inevitable when given a particular CPU architecture (e.g., x86). XMHF encapsulates this common functionality in a framework that allows others to build custom hypervisor-based solutions while freeing them from a considerable amount of (low-level, challenging to debug) wheel-reinventing that is often associated with such efforts.

We have ported a number of hypervisor research efforts to this new platform, essentially realizing a "version 2.0" implementation where development continues today. In this paper, we describe the design and implementation of XMHF, emphasizing those design decisions which we feel are suitable for supporting modular development of future "hypervisor applications" or "hypapps". We detail several case studies to substantiate these claims.

XMHF advocates a "rich" single-guest model where the hypervisor framework supports only a single-guest and allows the guest direct access to all performance-critical system devices and device interrupts. The single-guest model results in a dramatically reduced hypervisor complexity (since all devices are directly controlled by the OS) and consequently TCB, while at the same time promising near-native guest performance.

A hypapp relies on XMHF for core platform functionality while extending the framework to implement a customized solution. As a small piece of software between the OS and the hardware, hypapps therefore enjoy a unique advantage in terms of balance between security and versatility. They also help reduce security sensitive developers concerns with respect to other malicious applications with the OS or OS vulnerabilities.

XMHF currently supports both Intel and AMD commodity x86 hardware virtualized platforms and can run Linux, Windows XP and Windows 2003 as unmodified guests with SMP support. XMHF imposes less than 10% overhead in the common case, and the current implementatation has a TCB of 6–13K SLoC depending on the extent of framework features used by a hypapp.

We are encouraged by the end result – a clean, barebones hypervisor framework with desirable performance characteristics and an architecture amenable to formal analysis.

Contributions. We design a developer-centric hypervisor framework which supports modular development of future "hypervisor applications" or "hypapps".We implement XMHF on both Intel and AMD commodity x86 hardware virtualized platforms. Our framework is capable of running unmodified legacy SMP capable OSes such as Windows and Linux. We present a comprehensive performance evaluation of XMHF and describe our efforts in porting several recent hypervisor-based research efforts to XMHF to showcase the framework efficacy.

*We gratefully acknowledge support from the US Army Research Office contract number W911 NF 10 C 0037, and from CyLab at Carnegie Mellon University.

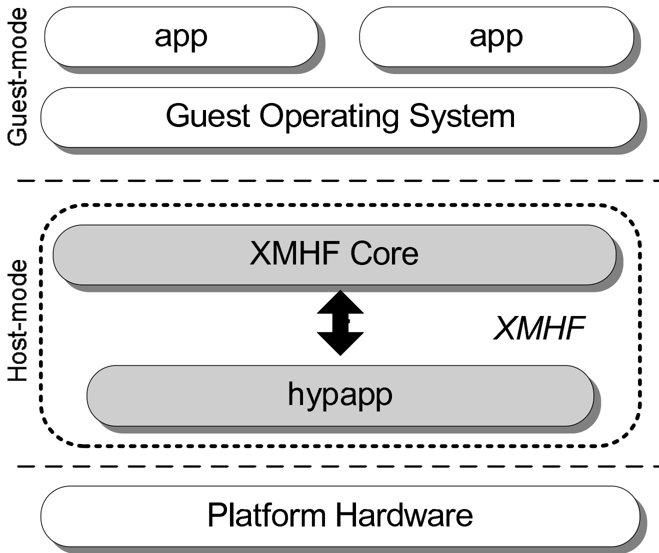

Figure 1: High level view of XMHF design

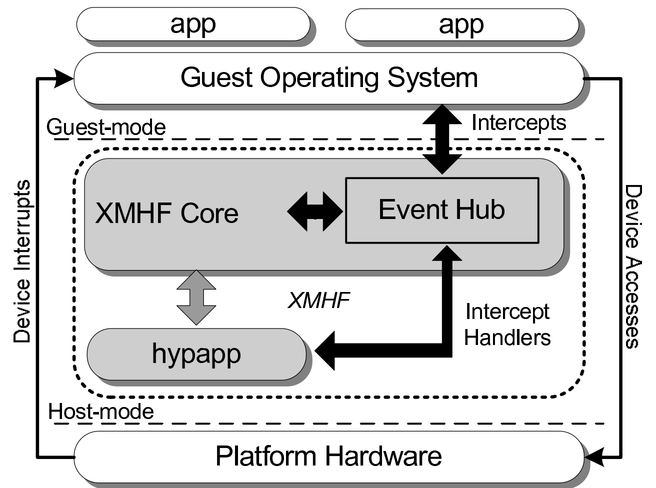

Figure 2: XMHF Rich Single-guest Execution Model

2. DESIGN

We design XMHF as a Type-1 (or native, bare metal) hypervisor that runs directly on the host's hardware to control the hardware and to manage guest operating systems. A guest operating system thus runs on another (deprivileged) level above the hypervisor. Our primary design choice for a bare-metal hypervisor is for a low-TCB and high performance hypervisor code base. Figure 1 shows the high-level design of XMHF. The XMHF framework consists of the XMHF core and supporting libraries that sit directly on top of the platform hardware. A hypapp extends XMHF and leverages the basic hypervisor and platform functionality provided by the XMHF core while implementing the desired (security) functionality.

2.1 "Rich" Single-guest Execution Model

We propose a "rich" single-guest execution model where the hypervisor framework supports only a single-guest and – following XMHF initialization (§2.2) – allows We note that the single-guest execution model resonates with a plethora of recent works [17, 24, 25, 29, 37, 38, 40, 41, 46] that attempt to provide desired functionality and security properties without depending on a commodity operating system. Note that the single-guest model allows its guest to be another (more traditional) hypervisor running multiple guest OSes, as evidenced by recent research efforts such as Cloudvisor [46] and Turtles [9].

The "rich" single-guest model (see Figure 2) has several advantages over traditional hypervisor approaches:

Dramatically reduced hypervisor complexity and consequently TCB. Since all devices are directly controlled by the guest, XMHF does not have to deal with per-device idiosyncrasies that arise from devices that are not completely standards-compliant. Further, XMHF does not need to perform hardware multiplexing, an inherently complex mechanism that can lead to security issues [16, 23]. This results in a small and simple hypervisor code base which improves maintainability and makes it amenable to formal verification and/or manual audits to rule out the incidence of vulnerabilities. Note that we could attempt to reduce device multiplexing by employing PCI-passthrough, where a device is directly assigned to a guest [3]. Unfortunately, the devices that can be allocated using this scheme depends directly on the bus hierarchy in a particular system [5]. Further, PCI-passthrough support is not seamless currently and requires non-trivial device firmware extraction and patching in some cases [4].

Narrow attacker interface. With the "rich" single guest execution model, the hypervisor interacts with the guest via a deterministic and well defined platform interface. For example, current x86 hardware virtualized platforms define a small deterministic set of *intercepts* that transfer control to the hypervisor upon detecting certain guest conditions (§ 3.3). This greatly reduces the attack surface of XMHF and the hypapp.

Near-native guest performance. The system interrupt controllers (IOAPICs) and devices are directly in the control of the guest. Therefore, all (device) interrupts are configured and handled by the guest without the intervention of XMHF. This results in a near-native guest performance (the guest still has to incur the memory/DMA protection overheads which are less than 5% in the common case (§ 5.3). This is in contrast to traditional hypervisors where the hypervisor virtualizes devices and interposes on all device interrupts. Note that even with PCI-passthrough, where a device can be directly assigned to a guest, the interrupts still need to be handled by the hypervisor resulting in noticable performance degradation in practice [20].

2.2 Isolation

As XMHF allows (in the common case) the guest to directly access system devices and handle interrupts, it must isolate itself from the guest in order to preserve its integrity – a fundamental hypervisor property. Integrity means that all changes to hypervisor memory are caused by direct action within the intended execution of the hypervisor's own instructions (e.g., initialization and inter- cept handlers). Further, integrity requires that neither hypervisor code nor data can be directly accessed via Direct Memory Accesses (DMA) by devices. Consequently, XMHF must ensure that

it starts up in an unmodified fashion and continues to run without any inadvertent modifications to its code and data.

We assume the adversary can execute arbitrary code within the guest and may also monitor and manipulate network traffic to and from the user's machine. However, we assume the adversary is remote and cannot perform physical attacks on the user's machine.

2.2.1 *Launch Integrity*

XMHF loads itself during the platform initialization and boot sequence. For XMHF to startup in an unmodified form entails trusting the platform initialization and boot sequence to correctly load and transfer control to XMHF. Unfortunately ensuring this is a challenge on commodity x86 platforms. The traditional x86 BIOS initialization and boot sequence is plagued by having existed for several decades. As such, modern security requirements and virtualization capabilities did not exist when it was first conceived. The result of this is that there may exist legacy code in a system's BIOS, option ROMs and boot-sequence that should not be trusted, since it was never subjected to rigorous analysis for security issues[†].

XMHF leverages dynamic root of trust (DRT) to startup in an unmodified fashion. A dynamic root of trust (DRT) is an execution environment created through a disruptive event that synchronizes and reinitializes all CPUs in the system to a known good state. It also disables all interrupt sources, DMA, and debugging access to the new environment. DRT support is available on all current commodity x86 CPUs from Intel and AMD [6, 22].

XMHF's launch process (see Figure 3) consists of an `init` module that is loaded via a (untrusted) boot-loader such as GRUB. The `init` module then uses appropriate CPU instructions to establish a DRT and loads the XMHF `secure-loader` in a memory constrained hardware protected environment. The XMHF `secure-loader` in turn initializes the platform and sets up required protections to run the XMHF `runtime` with full access to platform resources.

2.2.2 *Runtime Integrity*

Once XMHF has started in an unmodified form as described previously, it must continue to run without any inadvertent modifications to its memory regions (code and data). As XMHF allows the guest direct access to system devices, intuitively memory protection from devices and guest code (running on the CPU) becomes crucial to preserving its integrity and enforcing isolation.

Devices such as USB, Firewire, Storage and Network devices can directly access physical memory via DMA, potentially bypassing the hypervisor. These devices can be programmed by an attacker to access any portion of the physical memory including those belonging to the hypervisor [10]. Malicious firmware on a device can also accomplish the same goal by replacing legitimate physical memory addresses passed to it with hypervisor physical memory regions.

XMHF leverages the platform I/O Memory Management Unit (IOMMU) protect its memory regions from direct access by devices. The IOMMU is the only system device

[†]A closed system where only known firmware is executed at boot-time, can be subjected to analysis and consequently trusted. However, most (if not all) x86 systems do not fall under this category.

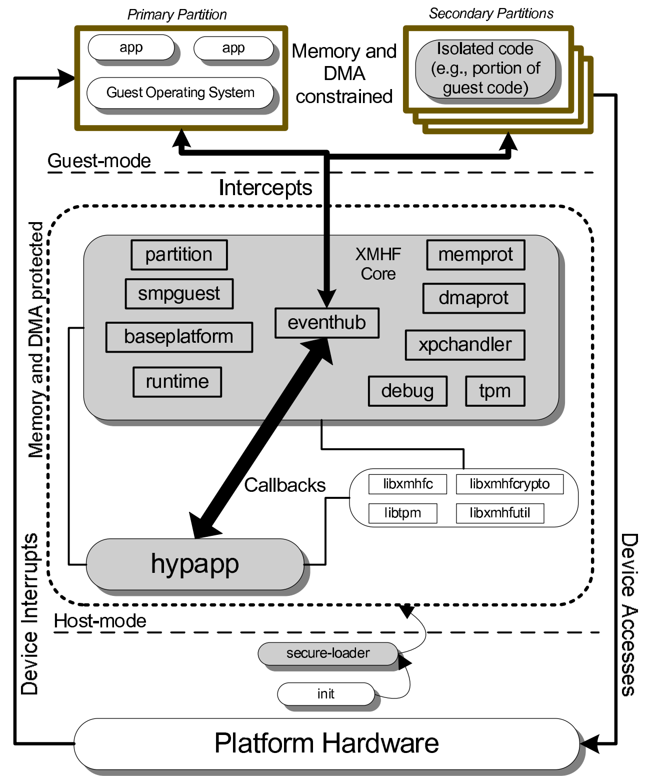

Figure 3: XMHF detailed architecture

that can intervene between DMA transactions occurring between a device and memory. Both AMD and Intel x86 platforms provide an IOMMU as a part of the northbridge. The IOMMU allows each peripheral device in the system to be assigned to a set of IO page tables. When an IO device attempts to access system memory, the IOMMU intercepts the access and uses the IO page tables associated with that device to determine whether the access is to be permitted as well as the actual location in system memory that is to be accessed. XMHF instantiates IO page tables such that physical addresses corresponding to the XMHF secure-loader and XMHF runtime memory regions are marked as inaccessible to any device.

A guest environment running on top of XMHF can run any commodity OS and applications. The guest may manipulate the CPU MMU's virtual memory data structures or even directly access system physical memory belonging to the hypervisor. XMHF uses *partitions*[‡] to contain guest code and data. Partitions in XMHF are essentially bare-bones CPU hardware backed execution containers which enforce system memory isolation for the guest or a portion of it.

XMHF creates a primary partition in order to run the guest operating environment. The primary partition allows the guest environment to see and access all the system devices including the CPU (and cores) just as it would have on a native boot. XMHF can also instantiate secondary

[‡]The term hardware virtual machine is used for such CPU execution containers in current hardware virtualization parlance. However, technically a virtual machine presents to the guest a virtualized view of the system devices in addition to enforcing memory isolation, and is a misnomer in our case.

partitions on demand when requested by a hypapp. These secondary partitions are capable of running portions of the guest environment code/data within isolation (Figure 3). This is useful when a hypapp wishes to implement desired security properties at a finer-granularity of portions of an untrusted application within the operating environment (e.g., TrustVisor [25])

2.3 Event-based Runtime Interface

Just like a regular application running inside an operating system accesses services via a plethora of interfaces (kernel, windowing, graphics, shell etc.), XMHF and hypapps interact with a guest operating system via an event-based interface during runtime. However, unlike regular application interfaces, this event-based interface is extremely small with well-defined CPU populated parameters, thereby greatly reducing the attack surface of XMHF and the hypapp. At the same time, this interface is versatile enough to realize several practical security applications as evidenced by several recent efforts [9,15,17,24–26,29,30,32,34,37,38,40,41,45,46].

XMHF leverages CPU support for hardware virtualization in order to capture and handle events within a guest operating environment. For example, current x86 hardware virtualized platforms define a deterministic set of *intercepts* that transfer control to the hypervisor upon detecting certain guest conditions (§ 3.3). Similar capabilities are also forthcoming in ARM processors [7]. XMHF, during initialization allows a hypapp to configure the set of guest events that it wishes to intercept and handle. This avoids unnecessary guest intercepts at runtime. The XMHF core `event-hub` gets control for all intercepted guest events and in turn invokes the appropriate XMHF/hypapp *callback* to handle the event. The XMHF/hypapp *callback* has the option of injecting the event back into the guest for further processing if desired. The event-callback mechanism therefore allows hypapp's to easily *extend* core XMHF functionality to realize desired functionality in the context of a particular guest.

2.4 Attested Measurements

Remote attestation allows remote parties to verify that a particular message originated from a particular software module. This is especially desirable in the context of a hypervisor framework such as XMHF since it forms the highest privilege system providing capabilities to enforce desired security policies.

Remotely ascertaining the cryptographic hash of executable code is commonly achieved via an attestation protocol. Such protocols generate a cryptographically signed or otherwise authenticated message containing one or more digests of the target executable code. We distinguish two facets of an attestation protocol: (1) accumulating measurements on the target system of interest, and (2) verifying the attested measurements on a trusted verifier system.

XMHF relies on a Trusted Platform Module (TPM) for accumulating integrity measurements. TPMs contain Platform Configuration Registers (PCRs), which are registers that store an append-only hash chain. Using dynamic root of trust as part of the launch process (§2.2.1) automatically extends the hash of the code being launched into a PCR. This PCR can be extended further as components are loaded and executed. XMHF `init` loads the XMHF `secure-loader`, which in turn measures and loads the XMHF `runtime`.

A challenge with this approach is that monitoring a PCR value for changes does not reveal any insight into the reason for a change. To be meaningful, remote attestations require a TCB that is relatively small and modular. It is then easier to keep track of a list of known-good components, and the hypervisor is presumed to be less susceptible to runtime compromise (due to its lower complexity / small size). This is important because the attestation is a load-time property (i.e., it only tells the verifier what code was loaded). Attestation cannot detect if a runtime exploit overwrites loaded code with unauthorized code.

XMHF addresses this problem by having a modular approach to its design (Figure 3). Every component of the core and supporting libraries can be essentially considered as objects exposing a set of interfaces and dealing with object data. Every XMHF component and the hypapp have the following memory layout: code, init-data, uninitialized-data and stack (if applicable). During load, the code and init-data are hashed and extended into a PCR for attestation purposes, while the uninitialized-data and stack are zeroed out to begin with.

3. IMPLEMENTATION

We now describe the XMHF implementation. The framework implementation supports both Intel and AMD commodity x86 hardware virtualized platforms and is capable of running unmodified legacy SMP capable Windows (2003 and XP) and Linux.

3.1 "Rich" Single Guest Execution Model

3.1.1 Multicore guest bringup

Summary. A native operating system, on the x86 platform uses the INIT and Startup Inter-Processor Interrupt (SIPI) in order to bring up multiple cores [6, 21]. The INIT-SIPI-SIPI sequence is delivered to a CPU core via the CPU Local Advanced Programmable Interrupt Controller (LAPIC). However when a guest OS runs on top of XMHF, the framework must ensure that it maintains required protections during runtime while allowing the guest OS to access the physical cores directly. Normal hypervisors such as Xen virtualize the CPU LAPIC in order to handle SMP guests. This results in considerable code complexity as a result of handling issues such as concurrency and interrupt ordering. In contrast, XMHF allows the guest direct access to the LAPIC. Therefore, ensuring that the framework is still able to retain control and maintain isolation during runtime is a challenge.
Details. On a single core CPU, XMHF, during initialization switches the Boot Strap Processor (BSP) core to guest-mode before booting the guest OS. For a multi core CPU XMHF, during initialization activates the remaining cores in the system and switches them to host-mode which then idle within XMHF. When the guest initiates the INIT and SIPI sequence in order to bring up multiple cores, XMHF intercepts this sequence and switches the cores to guest-mode before handing back execution to the guest.

To support both Intel and AMD x86 platforms, XMHF uses a unified scheme to intercept guest multi-core activation. The LAPIC Interrupt Control Register (ICR) is used to deliver the INIT-SIPI-SIPI sequence to a target core. On both Intel and AMD hardware-virtualized platforms, the LAPIC registers are accessed via memory-mapped I/O. The memory-mapped I/O region encompasses a single physical memory page. XMHF leverages hardware page tables to

to trap and intercept any changes to the LAPIC memory-mapped I/O page by the guest.

Subsequently, any writes to the LAPIC ICR by the guest causes the hardware to trigger an intercept. The XMHF core handles this intercept, disables guest interrupts and sets the guest *trap-flag* and resumes the guest. This causes the hardware to immediately trigger a single-step intercept, which is then handled by the XMHF core to process the instruction that caused the write to the LAPIC ICR. If a INIT command was being written to the ICR, XMHF simply voids the instruction. When the SIPI command is written to the ICR, XMHF voids the instruction and instead runs the target guest code on that core in *guest-mode*.

3.1.2 Core quiescing

Summary. In a multi-core system, the other cores need to be in a halted state when the XMHF core or hypapp is modifying critical data structures such as the nested page tables as this affects the way in which all cores perceive memory.

Detail. XMHF uses a mechanism called *core quiescing* in order to stall all cores in a multi-core system within a hypapp. XMHF leverages the Non-Maskable Interrupt (NMI) for core quiescing as described below. When a hypapp on a specific core C wants to perform quiescing it invokes the XMHF core interface which sends a NMI to all cores other than C. Since the NMI cannot be masked, this causes all these other cores to either receive a NMI intercept (if operating in guest mode) or a NMI exception (if operating in host mode). The NMI handler is invoked in both cases which is an idle spin-lock loop, in effect, stalling the core. Once the hypapp is done performing the required task on C, it signals the spin-lock which causes the other cores to resume.

3.1.3 Prevent access to critical system devices

Summary. Critical system devices such as the IOMMU and the system memory controller, like any other device expose their interface through either legacy IO or memory-mapped IO. For example, Intel x86 platforms expose the IOMMU as a DMA device through ACPI while AMD x86 platforms expose the DMA protection unit (DEV) as a PCI device. The system memory controller is typically exposed as a PCI device as well. With the "rich" single-guest model, the guest can perform direct I/O to these devices, effectively compromising the memory and DMA protections. Normal hypervisors such as Xen use a virtual BIOS to present to the guest a different platform device configuration and I/O areas and can easily mask platform critical devices from guest access. However, since XMHF lets the guest see and interact with the real system BIOS, it must employ a different mechanism to prevent access to platform critical system devices.

Detail. XMHF marks the ACPI and PCI configuration space of critical system devices (IOMMU, DEV and system memory controller for example) as not-present using the hardware page tables. Thus, the guest never gets to see these devices in the first place. It also makes the memory mapped I/O space of these devices inaccessible and sets intercepts on the legacy I/O space to prevent a guest from maliciously accessing these system devices.

3.1.4 Guest memory reporting

Summary. A normal guest operating system during its bootup uses the BIOS to determine the amount of physical memory in the system. More specifically the `INT 15h E820` interrupt interface is the standard way of obtaining the system physical memory map from the BIOS. However, with a hypervisor loaded, there must be a mechanism to report a reduced memory map devoid of the hypervisor memory regions to the guest. If not, the guest at some point during its initialization will end up accessing the hypervisor memory areas. As described previously, a traditional hypervisor solution maps a virtual BIOS for a virtual machine which reports a reduced system memory map. However, this mechanism cannot be used for the "rich" single-guest model where the guest gets to see and interact with the real system BIOS.

Detail. XMHF leverages the hardware page tables for the purpose of reporting a custom system memory map to the guest OS. During initialization XMHF locates the `INT 15h` interrupt handler address by scanning the real-mode Interrupt Vector Table (IVT). It then replaces the physical page entry corresponding to the interrupt handler in the nested page table to point to a buffer within XMHF. This buffer is an exact copy of the original interrupt handler page with the starting of the `INT 15h` handler replaced by a hypercall instruction. When the guest OS invokes the `INT 15h` service during system bootup, the hypercall instruction transfers control to XMHF which then checks to see if is it a memory map request. If so, it presents a custom memory map devoid of XMHF memory regions and resumes the guest. If not, it injects the interrupt back to the guest for further processing.

3.2 Isolation

3.2.1 Launch Integrity

XMHF's `init` module is responsible for initiating a dynamic root of trust (DRT) in order to bootstrap the XMHF `secure-loader`. DRT implementation has significant requirements on each of AMD and Intel x86 hardware platforms. We first briefly outline these requirements before proceeding to describe the XMHF `secure-loader` implementation. AMD calls the launched environment an Secure Loader Block (SLB). Intel calls the launched environment a Measured Launched Environment (MLE). For brevity we will use the term `secure-loader`.

XMHF `init`. On AMD platforms, XMHF `init` uses the *SKINIT* CPU instruction in order to establish a DRT and load the XMHF `secure-loader`. As per *SKINIT* requirements, XMHF `init` clears microcode on the BSP and all APs and ensures that the XMHF `secure-loader` is loaded on a 64K-aligned physical address.

Intel platforms support DRT with their Trusted eXecution Technology (TXT). As part of TXT several MSRs are added to the CPU which are used to setup required data structures for DRT initialization and are also used for status and error reporting. TXT also requires a chipset specific Authenticated Code Module (also known as SINIT AC module) to establish a DRT. XMHF `init` copies the appropriate SINIT AC module to the physical address specified by the TXT MSRs. The XMHF `init` module then constructs page tables that map the XMHF `secure-loader` so that the SINIT AC module can address it properly. These page tables are part of the XMHF `secure-loader` memory[§]. The XMHF `init` then sets up the CPU Memory Type Range Registers (MTRRs) in order to disable memory caching for the

[§]In practice 3 4K pages should suffice. These are PAE-formatted page tables.

SINIT AC module to run. Finally, XMHF `init` issues the GETSEC[SENTER] CPU instruction to establish a DRT and transfer control to the XMHF `secure-loader`.

XMHF `secure-loader`. Both Intel and AMD differ in their SLB/MLE size constraints, CPU state when `secure-loader` gets control and layout of the `secure-loader`. AMD SLBs are limited to 64K in size while Intel MLEs can be larger. Intel and AMD SLB/MLEs both start in 32-bit flat protected mode, but have different CPU states. On AMD, only CS and SS are valid. EAX contains the SL base address and EDX contains the CPU information. ESP is initialized to top of 64K. On Intel, CS and DS are valid and SS is invalid and uninitialized.[¶] Finally AMD SLBs and Intel MLEs have different header requirements.

XMHF uses a single 64KB `secure-loader` for both Intel and AMD platforms. The XMHF `secure-loader` memory image starts with 3 empty pages, except for the first 4 bytes. These are initialized to contain the SLB header for an AMD system. The entry point points beyond these three pages to the true entry point on the fourth page. On an Intel system these three pages will be overwritten with the MLE page tables by XMHF `init` as described previously. The MLE header is written into the MLE at the beginning of the fourth page, and serves no purpose when the `secure-loader` executes on an AMD platform. This scheme enables the XMHF `secure-loader` to meet both AMD and Intel DRT requirements with a single build process.

Because the XMHF `secure-loader` can be loaded anywhere in memory depending on the system memory map, the XMHF `secure-loader` needs to obtain its load address during runtime in order to setup required code, data and stack segments. While AMD platforms provide the base address in the EAX register, not all Intel platforms provide this information [‖]. XMHF `secure-loader` therfore uses a cross-processor solution to read and align the instruction pointer register to discover the base address. More specifically it uses the sequence: `call 1f; 1: popl %eax; andl $0xffff0000, %eax` which returns the runtime base address of the `secure-loader` in the EAX register.

Early DMA Protection. The XMHF `secure-loader` is loaded via a DRT operation which automatically records a cryptographic hash of the `secure-loader` in the platform TPM. The `secure-loader` in turn measures the XMHF runtime (including the core and the hypapp) and extends this measurement into the platform TPM before transferring control to the runtime. While DRT protects the XMHF `secure-loader` memory from the entire platform, the XMHF runtime is still vulnerable to DMA attacks from possibly malicious platform peripherals; A malicious peripheral can overwrite the runtime memory after the runtime is measured but before control is transferred to the runtime. To prevent such attacks, the XMHF `secure-loader` employs what we term "early" DMA protection as described below.

On the Intel platforms, the XMHF `secure-loader` employs the platform IOMMU hardware in order to setup DMA protection over the XMHF runtime code before measuring the runtime. The IOMMU hardware includes a set of protected memory registers (`PLMBASE_REG`, `PLMLIMIT_REG` and `PMEN_REG`) which can be programmed to contain the base address and limit of physical memory and turn on DMA protection for the specified range. XMHF `secure-loader` programs these registers to include the XMHF `secure-loader` and the XMHF runtime.

On AMD platforms, the XMHF `secure-loader` employs the Device Exclusion Vector (DEV) to setup early DMA protection. The DEV is a bitmap structure with each bit corresponding to a physical memory page in the system. If a bit is 1 then DMA is disallowed, otherwise DMA is allowed for the corresponding memory page. However, since the XMHF `secure-loader` and XMHF runtime are loaded at the top of system memory below 4GB, this presents an interesting problem: To be able to set DMA protections on the runtime we typically require an array of size $>$ 64KB. However, the size of the XMHF `secure-loader` is limited to 64KB in total (code and data). XMHF `secure-loader` circumvents this problem by using a protected 8K buffer (corresponding to a maximum 128MB runtime size) within its memory image. XMHF `secure-loader` then computes the DEV bitmap base address depending on the physical base address of the protected buffer and the physical base address of the memory region where the `secure-loader` was loaded. This aligns the 8K protected buffer to cover the `secure-loader` and runtime memory range upto a maximum of 128MB. XMHF `secure-loader` then sets the entire 8K buffer bits to 1's thereby DMA protecting the runtime. Note that we don't care about DMA protecting other parts of the DEV bitmap except for our protected buffer which is already DMA protected since the `secure-loader` was started using a DRT operation.

3.2.2 *Runtime Integrity*

Memory Isolation. XMHF uses two-level Hardware Page Tables (HPT)[**] for efficient memory isolation. In particular, the hardware ensures that all memory accesses by guest instructions go via a two-level translation in the presence of the HPT. First, the virtual address supplied by the guest is translated to a guest physical addresses using guest paging structures. Next, the guest physical addresses are translated into the actual system physical addresses using the permissions specified within the HPT. If the access requested by the guest violates the permissions stored in the HPT, the hardware triggers an intercept that can be handled by the XMHF core and/or the hypapp.

In XMHF, both primary (where the guest OS runs) and secondary partitions (recall § 2.2.2) are tied to a given HPT that enforces memory isolation. The partitions can all share the same HPT (for uniform view of memory and protections) or can have seperate HPTs (cases where the secondary partition might want to run only with a subset of the primary partition address space).

Runtime DMA Protection. XMHF `secure-loader` provides early DMA protection to the XMHF runtime. However, this protection is very coarse grained (over a single contiguous range). To provide hypapps with a more flexible and fine-grained DMA protection capability, the XMHF runtime reinitializes the DMA protection upon getting control. We now describe how fine-grained DMA protection is achieved on both AMD and Intel x86 platforms.

[¶]In practice, we have observed that SS still points inside the SINIT code region. Still, it is prudent not to depend upon this behavior.

[‖]If GETSEC[CAPABILITIES] indicates that ECX will contain the MLE base address pointer upon entry into the MLE, we can use ECX as the base address on Intel systems

[**]Called Nested Page Tables on AMD and Extended Page Table (EPT) on Intel Platforms respectively

Event-class	Callback Prototype	Callback handled by
CPU Control register access	`xmhf_hypappcb_handleCRaccess(vcpu, [CRparams...])`	XMHF core and hypapp
CPU Debug register access	`xmhf_hypappcb_handleDRaccess(vcpu, [DRparams...])`	XMHF hypapp
CPU MSR access	`xmhf_hypappcb_handleMSRaccess(vcpu, [MSRparams...])`	XMHF hypapp
CPU Exceptions	`xmhf_hypappcb_handleEXCP(vcpu, [EXCPparams...])`	XMHF hypapp
Non-maskable Interrupt (NMI)	`xmhf_hypappcb_handleNMI(vcpu, [NMIparams...])`	XMHF core
INIT/Shutdown	`xmhf_hypappcb_handleINIT(vcpu, [INITparams...])`	XMHF core and hypapp
CPU Descriptor Table access	`xmhf_hypappcb_handleXTRaccess(vcpu, [XTRparams...])`	XMHF hypapp
CPU Instructions	`xmhf_hypappcb_handleINSN(vcpu, [INSNparams...])`	XMHF core and hypapp
Hypercall	`xmhf_hypappcb_handleHYPC(vcpu, [HYPCparams...])`	XMHF hypapp
Legacy I/O access	`xmhf_hypappcb_handleIOaccess(vcpu, [IOparams...])`	XMHF core and hypapp
CPU Task Switch	`xmhf_hypappcb_handleTASK(vcpu, [TASKparams...])`	XMHF hypapp
CPU Nested Page Fault	`xmhf_hypappcb_handleNPF(vcpu, [NPFparams...])`	XMHF core and hypapp

Figure 4: XMHF uses an event-callback approach to allow the hypapp to interact with the guest. A hypapp can configure the primary and/or the secondary partitions to intercept required guest events choosing from the broad event classes on both AMD and Intel x86 platforms. For each event class, a hypapp callback is invoked in the corresponding CPU core context with associated parameters.

On AMD platforms, XMHF relies on the DEV for fine-grained DMA protection. As discussed previously, DEV's bitmap structure allows DMA protection to be set at a page granularity. On Intel platforms, XMHF uses the IOMMU page tables in order to provide fine-grained page-level DMA protection. The IOMMU has a master table called the Root-Entry-Table (RET) which is 4KB. Each 128-bit RET entry essentially corresponds to a PCI bus number (256 in total). Each RET entry points to a Context Entry Table (CET) which is 4KB in size. Each 128-bit CET entry accounts for 32 devices with 8 functions per device as per the PCI specification. Each CET entry points to a regular PAE-page table structure which contains mappings and protections for the DMA address space as seen by the device. Since XMHF uses the "rich" single-guest model, all CET entries point to a single PAE-page table structure that controls the DMA protections for all devices in the system.

3.3 Event-based handling

As mentioned in §2.3 XMHF uses an event-callback approach to allow the hypapp to interact with the guest and provide required capabilities. A hypapp can configure the primary and/or the secondary partitions to intercept required guest events choosing from the broad event classes on both AMD and Intel x86 platforms as shown in Figure 4. For each intercepted class of event the XMHF `event-hub` component invokes hypapp callback with the associated parameters in the context of the physical CPU core on which the intercept was triggered. For example, on a nested page fault, the hypapp gets the faulting virtual address, physical address and the error code associated with the fault. Note that the XMHF `event-hub` hides sub-architecture (AMD vs Intel) specific details and presents the hyperapp with a common architectural (x86) state. The hypapp callback is of course free to include sub-architecture specific handling as needed. The hypapp callback can choose to handle the event and/or inject it back to the guest for further processing. Also note from the figure that some events are also processed by the core directly, e.g., to handle core-quiescing (NMI intercept), shutdown (INIT/shutdown) and SMP guest bringup (INIT, DB exception and nested page fault intercept).

3.4 Attested Measurements

We first summarize the memory layout of the XMHF `secure-loader` and the XMHF runtime. We then describe how hashes of various components are computed during the build process. Finally we describe how these hashes are then used to attest the launch of XMHF and the hypapp to a local or remote verifier.

The memory layout of the XMHF runtime components consist of the code, init-data, uninit-data and stack as described previously in §2.4. The XMHF `secure-loader` contains an extra region called `.sl_untrusted_params` which contains the untrusted input data for the `secure-loader` that is passed by XMHF `init`. This untrusted data is a `secure-loader` parameter block which contains boot time parameters that are used by the XMHF core for platform initialization and configuration information that may be hypapp specific. The XMHF `secure-loader` is responsible for validating all input data in the `.sl_untrusted_params` region. It is not measured by default, though a particular hypapp may measure anything it likes.

During the hypapp build process, the XMHF runtime components and the XMHF `secure-loader` are built and the expected ("golden") hash itself is computed from the XMHF `secure-loader` and runtime binaries using a utility such as `sha1sum`. This "golden" hash can be used to compare the result of a platform attestation to convince a local or remote verifier that XMHF and the hypapp executed on the platform. This process is further described in detail below.

The XMHF `secure-loader` is measured by the CPU during creation of the DRT (i.e., SKINIT on AMD or GETSEC[SENTER] on Intel). The first 64K of the secure loader are measured implicitly during DRTM establishment and stored in TPM PCRs 17 or 18 on AMD and Intel respectively. Note that the expected measurement of the `secure-loader` will be different depending on whether it was was launched on an Intel CPU or an AMD CPU due to Intel's requirement for an SINIT module and the differences in the `secure-loader` header as described previously. The `secure-loader` in turn hashes the XMHF runtime and extends its measurement into PCR 17 (or 18). Thus, PCR 17 (or 18) will take on the value:
$H(H(0x00||H(\texttt{secure-loader}))||H(runtime))$. The properties of the hash function, TPM, chipset, and CPU guarantee that no other operation can cause PCR 17 (or 18)

to take on this value. Thus, an attestation of the value of PCR 17 (or 18) will convince the verifier that XMHF and the intended hypapp was launched on the target platform.

3.4.1 TPM sharing

XMHF leverages the TPM for platform attestation. However, XMHF also lets the guest environment see the physical TPM and communicate with it directly. Thus, XMHF has to multiplex the TPM in a way so that both the guest and XMHF and/or the hypapp can access the TPM functionality efficiently and without disruption. XMHF leverages TPM localities for this purpose [36].

The TPM definition provides for localities or indications of specific platform processes. The TPM PC Specific Specification [35] defines five localities 0 through 4. A reserved set of page-aligned memory addresses correlates with the localities. Locality 4 is used by the hardware DRT mechanism and allows the TPM to respond appropriately to hardware DRT requests (i.e, SKINIT and GETSEC[SENTER]) for measurement and resetting of PCR. Locality 3 is provided for (optional) auxillary components that may or may not be part of the hardware DRT process. XMHF reserves locality 2 for the hypapp and also uses it to store the `secure-loader` and runtime measurements. XMHF reserves localities 0 and 1 for use by the guest. Futher, XMHF also masks locality 2 from the view of the guest by using nested page tables to mark the corresponding locality 2 memory page as not-present. This restricts the guest to use only TPM locality 0 and 1 for its desired purpose.

4. DEVELOPER'S PERSPECTIVE

In this section, we summarize the process of writing, building, deploying, and debugging a hypapp from the perspective of a hypapp developer.

Writing a hypapp . We have developed XMHF primarily in C, with a small portion of the core functionality written in assembly. Every hypapp contains a `hypapp_main` function which gets control in the context of each physical CPU core when XMHF initializes. This allows the hypapp to perform any one-time state initialization before the guest environment is loaded. During guest runtime, in response to an intercepted event the appropriate hypapp callback (see Figure 4) is invoked by the XMHF core. When the guest finally shuts down, the hypapp shutdown callback gets control, which can perform any required cleanup.

Building a hypapp . To convert a hypapp into a final binary, we link it against the XMHF core component library (representing XMHF core functionality). The build process automatically generates the `init` binary and a combined binary image of the `secure-loader`, the XMHF core and runtime, and the hypapp.

Regular application developers depend on a variety of OS kernel and supporting libraries. There is no reason this should be any different in the case of a hypapp developer, except that it is desirable to modularize the libraries further than is traditionally done to help minimize the library interface and the amount of code included in the hypervisor's TCB. We have developed several small libraries in the course of applying XMHF to develop hypapps described in §5.2. The following paragraphs provide a brief description of these libraries.

XMHF core: The XMHF core provides the minimal functionality needed to support hypapps. In brief, the XMHF core contains logic to invoke the hypapp initialization and shutdown routines in response to a platform initialization sequence and guest shutdown. The XMHF core also provides interfaces for core quiescing and resumption in case of SMP guests. Finally, the XMHF core invokes various hypapp callbacks (if any are defined) in response to various guest events allowing the hypapp to implement desired functionality in the context of the guest.

libxmhfc: We have implemented a tiny version of a C9x compatible C runtime library for use by the XMHF core and hypapps. This includes basic string (strncpy, etc.), memory (memcpy, memmove, etc.) and standard I/O (printf, etc.) functionality.

libxmhfcrypto: We have developed a small library of cryptographic functions. Supported operations include RSA key generation, RSA encryp- tion and decryption, SHA-1, HMAC and MD5.

libtpm: The TPM library allows the XMHF core and hypapps to perform useful TPM operations. Currently supported operations include GetCapability, PCR Read, PCR Extend, GetRandom, NV Read, and NV Write.

libxmhfutil: Optional general utility functions such as hardware page table abstractions for masking AMD NPT and Intel EPT differences and for handling various guest paging modes (such as non-PAE and PAE), and command line parsing functions to parse any boot time options that a hypapp may wish to process.

Deploying a hypapp . XMHF currently loads during system startup via a boot-loader such as GRUB. Note that the Intel platforms require an addition SINIT AC module to be passed via the boot-loader for DRT establishment (§ 3.2.1).

Debugging a hypapp . Debugging is an important aspect of normal application development. More so in the case of a hypapp which is executing with hypervisor privileges and does not have access to regular interactive debugging tools. Many development issues that can be easily identified in a userspace application tend to be harder to pinpoint within a hypervisor simply because of a lack of suitable debugging interface. XMHF currently provides hypapp console support via serial I/O. This can be currently leveraged to trace/debug hypapp execution during runtime in a non-disruptive manner.

5. EVALUATION

We present the TCB size of XMHF's current implementation and describe our efforts in porting several recent hypervisor-based research efforts as hypapps running on XMHF. We then present the performance impact on a legacy guest operating system running on XMHF and evaluate the performance overhead that XMHF imposes on a hypapp. Finally, we compare XMHF's performance with the popular open-source Xen hypervisor. These results explain the basic hardware virtualization overhead intrinsic to the design of XMHF.

5.1 Trusted Computing Base (TCB)

Figure 5 shows XMHF's TCB. The framework TCB can be split into XMHF core and the supporting libraries (`libxmhfc`, `libxmhfcrypto`, `libxmhfutil` and `libtpm`). From the hypapp's perspective, the minimum TCB exposed by XMHF is 6683 SLoC. This consists of the entire XMHF core, `libxmhfc` and a couple of functions from `libtpm` (`TPM_extend`) and `libxmhfcrypto` (`sha1`). The full TCB for XMHF is 13555

Component	ASM	C	.h	Full TCB	Min. TCB
XMHF core	717	5036	3987	5753	5753
libxmhfc	0	639	1349	639	639
libxmhfcrypto	0	2382	835	2382	218
libxmhfutil	0	3837	913	3837	0
libtpm	0	944	382	944	73
Total				13555	6683

Figure 5: XMHF's Trusted Computing Base in Source Lines of Code (SLoC)

	Native	XMHF-4K	XMHF-2M
Latency in micro-seconds (smaller is better)			
null syscall	0.05	0.05	0.05
fork	263	458	317
exec	672	1357	937
ctxsw	31.8	38.1	39
mmap	4672	14200	4868
socket	21.1	28.8	23.6
Bandwidth in MB/sec (bigger is better)			
mem read	5187	5170	5256
mem write	5433	5344	5440

Figure 7: XMHF's lmbench OS microbenchmarks

SLoC including every function in all supporting libraries.

5.2 hypapp Case Studies

We demonstrate the utility of XMHF in the context of porting several recent research efforts in the hypervisor space to XMHF. We choose TrustVisor [25], Lockdown [37], XTRec [38], SecVisor [29] and HyperDbg [17] as our candidate hypapps due to the availability of their sources. Figure 6 shows the SLoC metrics and platform support for each hypapp before and after the port to XMHF. Note that the SecVisor, XTRec and HyperDbg ports are still work in progress. We derived the SLoC metrics for these by manual inspection of their existing sources and distinguishing between the hypervisor core and application specific logic. As seen, the XMHF core and support libraries form over 60% of the hypapp's TCB on average. This supports our hypothesis that these hypervisors share a common code base that is re-used or engineered from scratch with every new application. Also, using XMHF endows the hypapps with both Intel and AMD x86 SMP platform support for free.

5.3 Performance Measurements

We measure XMHF's runtime performance using two metrics: (a) guest overhead imposed solely by the framework (i.e., without a specific hypapp), and (b) base overhead imposed by XMHF for any given hypapp.

Our experimental platform is a HP Elitebook 8540p with a Quad-Core Intel Core i7 running at 3 GHz, 4 GB RAM, 320GB SATA HDD and an an Intel e1000 ethernet controller. We use Ubuntu 12.04 LTS as our guest operating system running the Linux kernel v3.2.2. For network benchmarks, we connect another machine via a 1 Gbps Ethernet crossover link and run the 8540p as a server. We use XMHF with both 4K and 2MB hardware page table (HPT) mappings for measurement purposes.

5.3.1 Guest Performance

XMHF only receives control as a result of a hypercall or guest event (recall § 2.3). Thus, when well-behaved legacy guest runs, the performance overhead is exclusively the result of the hardware virtualization mechanisms, particularly the hardware nested paging.

OS Microbenchmarks. We use the lmbench suite to measure the overhead of different OS operations when running on top of XMHF. Figure 7 shows the results of 8 important operations in our experiments: null systemcall, fork, exec, ctxsw (context switch among 16 processes, each 64 KB in size), mmap, socket (local communication by socket), mem read (memory read bandwidth) and mem write (memory write bandwidth). We compare the native system and XMHF (with both 4K and 2MB hardware page table configurations). Most of the benchmarks run with acceptable overheads. However, fork, exec and mmap run with comparatively higher overheads. This is due to the fact that those operations stress the system's MMU and TLB functionality – components which are highly sensitive to the hardware performance of nested page tables. Also, in general, the overhead is larger for a 4K HPT configuration when compared to a 2MB HPT configuration. This is due to fewer TLB mappings in the latter case leading to better TLB utilization. We note that these overheads are likely to decrease on future platforms as hardware virtualization support matures.

Application Benchmarks. We execute both compute-bound and I/O-bound applications with XMHF. For compute-bound applications, we use the SPECint 2006 suite. For I/O-bound applications, we use the iozone (disk read and write) and compilebench (project compilation) benchmarks from the open-source phoronix-test-suite[††], , and unmodified Apache web server performance.

We configure iozone with 4K block size and 2GB file size and perform the disk read and write benchmarks. We choose the compile benchmark from compilebench. We run the Apache web server on the system running XMHF, and use the Apache Benchmark (ab) included in the Apache distribution to perform 200,000 transactions with 20 concurrent connections.

Our results are presented in Figure 8. Most of the SPEC benchmarks show less than 3% performance overhead. However, there are four benchmarks with over 10%, and two more with 20% and 55% overhead. We attribute this high overhead to paging operations performed with the current HPT, and expect that performance will improve as HPT hardware matures. For I/O application benchmarks, read access to very large files and network incurs the highest overhead (40% and 25% respectively). The rest of the benchmarks show less than 10% overhead. We also expect this overhead to diminish with newer HPT hardware. In general, for both compute and I/O benchmarks, XMHF with 2MB HPT configuration performs better than XMHF with 4KB HPT configuration.

5.3.2 Performance of hypapps

A hypapp built on top of XMHF incurs two basic overheads during execution: (a) each time the hypapp is invoked via intercepted guest events (including a hypercall), and (b) each time the hypapp quiesces cores in a multi-core system

[††] `http://phoronix.com`

	Original			On XMHF					
hypapp	SLoC	Arch. Support	SMP Support	XMHF SLoC	hypapp SLoC	Total SLoC	% XMHF SLoC	Arch. Support	SMP Support
TrustVisor	6481	x86 AMD	No	10365	4049	14414	72%	x86 AMD, Intel	Yes
Lockdown	10KLOC	x86 AMD	No	6683	8416	15099	44%	x86 AMD, Intel	Yes
XTRec	2195	x86 AMD	No	6683	1506	8189	82%	x86 AMD, Intel	Yes
SecVisor	1760	x86 AMD	No	6683	1289	7972	84%	x86 AMD, Intel	Yes
HyperDbg	18967	x86 Intel	No	6683	17190	23873	28%	x86 AMD, Intel	Yes

Figure 6: Porting status of several hypervisor-based research efforts as XMHF hypapps

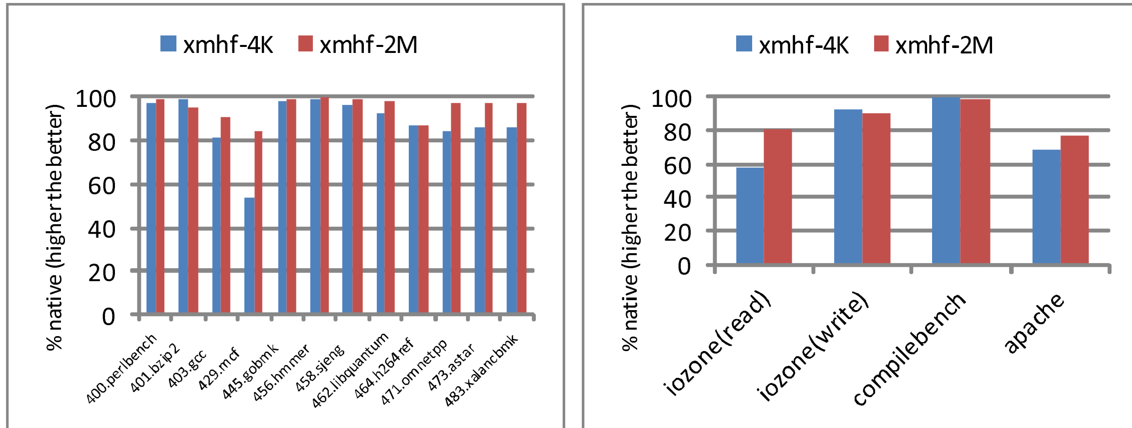

Figure 8: XMHF Application Benchmarks

	Event Trigger	Core Quiescing
XMHF (4K)	10.12	13.58
XMHF (2M)	10.00	14.34

Figure 9: XMHF hypapp overhead microbenchmarks (in micro-seconds). Avg. of 100 runs with negligible variance.

in order to perform hardware page table (HPT) updates.

Each time the hypapp is invoked, the CPU must switch from guest mode to host mode, which includes saving the current guest environment state and loading the host environment state. After the hypapp finishes its task, the CPU will switch back to the guest by performing the reverse environment saving and loading. Thus, there is a performance impact from cache and TLB activity. We measure this overhead by invoking a simple hypercall within the guest and measuring the round-trip time. As described in § 3.1.2, when a hypapp is modifying HPTs in a multi-core system, it must quiesce the other cores before the modification and then release them once the operation has been performed. As XMHF uses the NMI for this purpose, it results in a measurable performance overhead. We measure the quiesce overhead by using a simple hypapp that quiesces the cores, performs a nop and releases them in response to a guest hypercall event. We use a guest application that invokes the hypercall and measure the rount-trip time.

Figure 9 shows the hypapp overheads on XMHF for both 4K and 2MB HPT configurations. As seen both the event trigger and quiescing overheads are minimal (10-13 microseconds). Also, these overheads seem independent of the HPT configuration employed.

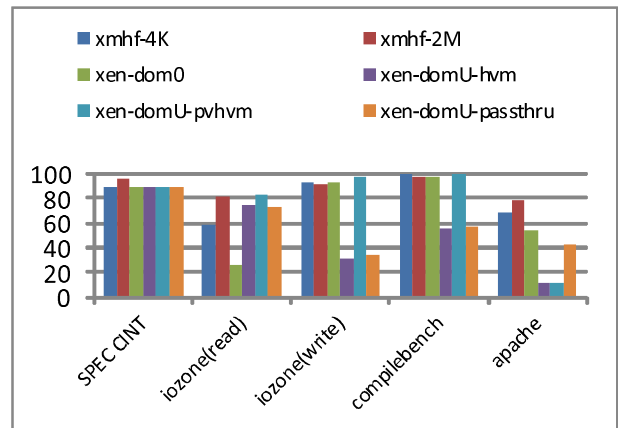

Figure 10: XMHF Performance Comparison with Xen

5.4 Performance Comparison

We now compare XMHF's performance with the popular Xen hypervisor. We use the latest version of Xen (v 4.1.2 as of this writing) with Ubuntu 12.04 LTS (kernel 3.2.2) for comparison purposes. We use three hardware virtual machine (HVM) configurations for domU, that are identical in memory and CPU configuration to the native system: HVM domU (xen-domU-hvm), HVM domU with paravirtualized drivers (xen-domU-pvhvm) and HVM domU with pci-passthrough (xen-domU-passthru). We also use dom0 (xen-dom0) as a candidate for performance evaluation. For compute bound applications we use the SPECint suite. For I/O-bound applications we use iozone(read and write), compilebench and apache with same parameters as described

previously in § 5.3.1.

Figure 10 shows our performance comparison results. For compute-bound applications XMHF and Xen have similar overheads (around 10% on average) with the 2MB XMHF HPT configuration performing slightly better. For disk-I/O benchmarks, XMHF, xen-dom0 and xen-domU-pvhvm have the lowest overheads (ranging from 3-20%). Both XMHF and xen have higher overheads on the disk read benchmark when compared to other disk benchmarks. For network-I/O benchmark, XMHF has the lowest overhead (20-30%). xen-dom0 and xen-domU-passthru incur a 45% and 60% overhead respectively, while xen-domU-hvm and xen-domU-pvhvm have more than 85% overhead.

6. DISCUSSION

We now discuss additional issues, including opportunities for formal verification of the XMHF framework.

6.1 Formal Verification

Datta et al. [14] show that support for dynamic root of trust is a viable means for building a system with code and execution integrity. We plan to build on the results of Datta et al. to prove the integrity property of the XMHF framework. We also plan to verify the XMHF implementation using software model checking methods [11,13].

6.2 Future Work and Optimizations

We have already identified several optimizations that are not implemented in XMHF currently but that will further reduce the overhead imposed by XMHF or increase its applicability.

The first is support for physical memory beyond 4GB. We are planning on leveraging XMHF's existing 32-bit execution environment and the x86 PAE paging mode in order to map and use physical memory beyond 4GB. A longer term goal is a full 64-bit transition.

XMHF and hypapps currently load during system bootup and in turn load and boot a guest OS on top. The framework then remains active during the entire guest OS lifetime. However, XMHF should have support for launching itself underneath an already running OS on-demand and should have support for unloadig itself when it is not needed.

In the current XMHF framework, hypapps execute at the highest privilege level like the XMHF core. While it is desirable to let the hypapp have unfettered access to the system, we plan on investigating light-weight de-privileging mechanisms that would efficiently isolate the hypapp from the XMHF core while allowing the hypapp full access to the system and XMHF core functionality.

Finally, XMHF should have support for recursive virtualizability, so that the framework does not monopolize the use of hardware virtualization features.

7. RELATED WORK

BitVisor [31] is a tiny open-source hypervisor designed for mediating I/O access from a single guest OS. It can allow a particular device to be access controlled by the hypervisor while allowing other devices to be handled by the guest directly. BitVisor has a size of 20K SLoC and supports a single guest VM running either Windows or Linux on the Intel x86 platform. XMHF, in the same spirit advocates the single "rich" guest model (§ 2.1). XMHF uses a dynamic root of trust for initialization and allows more fine-grained interception of guest events. The framework extensibility allows a wide range of hypapps to be built around it. XMHF supports multiprocessor configuration on both AMD and Intel x86 platforms.

Xen [8], KVM [27], VMware [39], NOVA [33], Virtualbox[‡‡] and L4 are some popular general purpose (open-source) hypervisors and microkernels which have been used for various hypervisor based research [9,15,26,30,32,34,45]. However, unlike XMHF, they do not present clear extensible interfaces for developers of hypervisor-based applications. Further, complexity arising from resource multiplexing and and increased TCB make them prone to various security vulnerabilities [1,2,42–44].

Qubes [28] is an open-source effort to build a secure desktop OS based on Xen. Qubes isolates various programs from each other, and even sandboxes many system-level components, like networking or storage subsystem using Xen virtual machines. It essentially leverages Xen's core functionality and can currently run Linux para-virtualized guests.

OsKit [18] is an early work on providing a framework for modular OS development. The OSKit makes it vastly easier to create a new OS on x86 platforms, port an existing OS to the x86 or enhance an OS to support a wider range of services. In some sense, XMHF attempts to provide the same kind of modular and extensible infrastructure for creating and porting new hypervisor-based applications on commodity x86 platforms.

8. CONCLUSIONS

In this paper, we propose an eXtensible and Modular Hypervisor Framework (XMHF) which strives to be a comprehensible and flexible platform for building hypervisor applications (hypapps). By providing in a modular way not only most of the infrastructure "grunge" needed by an hypervisor application, but also supporting libraries, XMHF's goal is to lower the barrier to develop new and exciting hypapps with a low TCB while being compatible with unmodified legacy operating systems and applications. Given XMHF's features and performance characteristics, we anticipate that it can significantly enhance hypervisor research and development.

9. AVAILABILITY

XMHF is open-source software and is available at the following URL:

`http://xmhf.org`

10. ACKNOWLEDGMENTS

We are grateful to Sagar Chaki, Anupam Datta, Virgil Gligor, Limin Jia, and Adrian Perrig for insightful discussions surrounding the design and architecture of XMHF. Zongwei Zhou and Yanlin Li identified many issues in the implementation.

11. REFERENCES

[1] Elevated privileges. CVE-2007-4993, 2007.

[2] Multiple integer overflows allow execution of arbitrary code. CVE-2007-5497, 2007.

[‡‡] http://virtualbox.org

[3] Xen pcipassthrough. http://wiki.xensource.com/xenwiki/XenPCIpassthrough, Oct. 2011.

[4] Xen vgapassthrough. http://wiki.xensource.com/xenwiki/XenVGAPassthrough, Oct. 2011.

[5] Xen vtdhowto. http://wiki.xensource.com/xenwiki/VTdHowTo, Oct. 2011.

[6] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24594 rev. 3.11, Dec. 2005.

[7] ARM Limited. Virtualization extensions architecture specification. `http://infocenter.arm.com`, Oct. 2010.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[9] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[10] A. Boileau. Hit by a bus: Physical access attacks with firewire. RuxCon, 2006.

[11] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[12] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of ASPLOS*, Mar. 2008.

[13] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[14] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[15] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, 2008.

[16] N. Elhage. Virtunoid: Breaking out of kvm. Defcon, 2011.

[17] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, 2010.

[18] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for os and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SOSP*, 2003.

[20] A. Gordon, M. Ben-Yehuda, N. Amit, N. HarÉl, A. Landau, and A. Schuster. ELI: Bare-Metal Performance for I/O Virtualization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2012.

[21] Intel Corporation. IA-32 Intel architecture software developer's manual. Intel Publication nos. 253665–253668, 2005.

[22] Intel Corporation. Trusted execution technology – preliminary architecture specification and enabling considerations. Document number 31516803, Nov. 2006.

[23] P. Karger and D. Safford. I/O for virtual machine monitors: Security and performance issues. *IEEE Security and Privacy*, 6(5):16–23, 2008.

[24] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, 2008.

[25] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[26] D. Quist, L. Liebrock, and J. Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2):121–131, May 2011.

[27] RedHat. KVM – kernel based virtual machine. `http://www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`, 2009.

[28] J. Rutkowska and R. Wojtczuk. Qubes os architecture. `http://qubes-os.org`, 2010.

[29] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, 2007.

[30] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, 2009.

[31] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[32] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[33] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[34] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[35] Trusted Computing Group. PC client specific TPM interface specification (TIS). Version 1.2, Revision

1.00, July 2005.

[36] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 103, 2007.

[37] 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 *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, June 2012.

[38] A. Vasudevan, N. Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 44th IEEE Hawaii International Conference in System Sciences (HICSS'11)*, Jan. 2011.

[39] VMware Corporation. VMware ESX, bare-metal hypervisor for virtual machines. `http://www.vmware.com/products/vi/esx/`, Nov. 2008.

[40] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[41] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, 2012.

[42] R. Wojtczuk. Detecting and preventing the Xen hypervisor subversions. Invisible Things Lab, 2008.

[43] R. Wojtczuk. Subverting the Xen hypervisor. Invisible Things Lab, 2008.

[44] R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. Invisible Things Lab, 2008.

[45] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[46] 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*, SOSP '11, 2011.