2013 IEEE Symposium on Security and Privacy

Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework*

Amit Vasudevan*, Sagar Chaki†, Limin Jia*, Jonathan McCune‡, James Newsome§ and Anupam Datta*
*CyLab, Carnegie Mellon University (amitvasudevan@acm.org, liminjia@cmu.edu, danupam@cmu.edu)
†SEI, Carnegie Mellon University (chaki@sei.cmu.edu)
‡Google Inc. (jonmccune@google.com)
§Independent Consultant (jim@jimnewsome.net)

***Abstract*— We present the design, implementation, and verification of XMHF– an eXtensible and Modular Hypervisor Framework. XMHF is designed to achieve three goals – *modular extensibility*, *automated verification*, and *high performance*. XMHF includes a core that provides functionality common to many hypervisor-based security architectures and supports extensions that augment the core with additional security or functional properties while preserving the fundamental hypervisor security property of *memory integrity* (i.e., ensuring that the hypervisor's memory is not modified by software running at a lower privilege level). We verify the memory integrity of the XMHF core – 6018 lines of code – using a combination of automated and manual techniques. The model checker CBMC automatically verifies 5208 lines of C code in about 80 seconds using less than 2GB of RAM. We manually audit the remaining 422 lines of C code and 388 lines of assembly language code that are stable and unlikely to change as development proceeds. Our experiments indicate that XMHF's performance is comparable to popular high-performance general-purpose hypervisors for the single guest that it supports.**

***Keywords*-Hypervisor Framework, Memory Integrity, Verification, Hypervisor Applications ("Hypapps")**

I. Introduction

Hypervisor-based architectures for improving system security have been extensively explored in recent years [1]–[16]. These systems are designed to provide interesting security and functional properties including secrecy of security sensitive application code and data [7], trusted user and application interfaces [2], [4], [13], application integrity and privacy [3], [5], [10], [11], [17], debugging support [8], malware analysis, detection and runtime monitoring [6], [9], [14]–[16] and trustworthy resource accounting [1]. A majority of these hypervisor-based solutions are designed and written from scratch with the primary goal of achieving a low Trusted Computing Base (TCB) while providing a specific security property and functionality in the context of an operating system or another (more traditional) hypervisor [2]–[10]. Other hypervisor-based approaches leverage existing general-purpose virtualization solutions (e.g., Xen, VMware, Linux KVM) for convenience, but generally don't require such functionality [1], [11], [13]–[17].

This paper describes the design, implementation and verification of an open-source eXtensible and Modular Hypervisor Framework (XMHF) that can serve as a platform for performing security-oriented hypervisor research and development. Observing that hypervisor-based security solutions often rely on a common core functionality given a particular CPU architecture, XMHF is designed to provide this core functionality while at the same time supporting extensions that can provide custom hypervisor-based solutions ("hypervisor applications" or "hypapps") for specific functional and security properties. The core of XMHF thus has a small TCB. All extensions reuse the core, avoiding the need to re-implement it correctly. Furthermore, the XMHF design enables automated verification of relevant security properties of its core and ensures that the properties are preserved as extensions (hypapps) are added as long as the extensions manipulate security-sensitive state using prescribed interfaces provided by the core. At the same time XMHF's performance is comparable to popular high-performance general-purpose hypervisors for the single guest that it supports.

XMHF supports a single full-featured commodity guest OS ("rich" guest). We make this design decision in order to achieve our design goals—*modular extensibility*, *automated verification*, and *high performance*. Specifically, XMHF leverages hardware virtualization primitives to allow the guest direct access to all performance-critical system devices and device interrupts. This model results in reduced hypervisor complexity (since all devices are directly controlled by the OS) and consequently TCB, as well as promising high guest performance (since device interrupts do not trap to the hypervisor). Further, the single-guest model allows XMHF to be designed for sequential execution (e.g., no interrupts within the hypervisor) while allowing the guest to use multiple CPUs, be multi-threaded and handle device interrupts. As a result, the automated component of our verification only requires model checking sequential programs, rendering it more tractable.

The focus of our verification efforts is *memory integrity*, a fundamental hypervisor security property. Roughly, memory integrity denotes that hypervisor memory regions can only be modified by instructions that are an intended part of

*XMHF is open-source software and is available at: **http://xmhf.org**

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.36

the hypervisor. Without memory integrity, portions of the hypervisor that manage the isolation of memory pages are open to malicious modifications, thereby allowing one guest to modify the code or data of another guest or the hypervisor itself. Memory integrity is therefore essential for realizing other important security goals of the hypervisor and hypapps, such as data secrecy and availability of the hypervisor as well as guests.

We call our design methodology DRIVE – "Designing hypervisors for Rigorous Integrity VErification". DRIVE is composed of a set of hypervisor properties and system invariants. The hypervisor properties entail the invariants, which in turn imply the hypervisor's memory integrity. Some of the properties and invariants are guaranteed by the hardware and the system architecture, while others are discharged via automated verification. DRIVE makes explicit which (properties and invariants) must be verified, which are assumed and which are guaranteed. Thus, DRIVE enables a synergy between architecture and automated analysis to ensure hypervisor memory integrity.

There have been several efforts to verify security-relevant properties of hypervisor systems [18]–[21]. However, these approaches rely on theorem proving, and are less automated. In addition, they are not focused on designing the target hypervisor in a way that reduces re-verification efforts upon changes to its implementation. XMHF is architected such that the portions requiring manual re-auditing are small, stable and unlikely to change as development proceeds. We emphasize automated verification of the portions of the XMHF code base that are subject to change as development proceeds, e.g., guest event handling and new hypapps. Compared to existing efforts, XMHF allows verification during its development. This fulfills the design goal of XMHF to serve as a framework on which developers build their specific hypapp(s) without sacrificing memory integrity.

The XMHF implementation currently supports both Intel and AMD x86 hardware virtualized platforms and is capable of running unmodified legacy multiprocessor capable OSes such as Windows and Linux. The XMHF core has a TCB of 6018 SLoC, and its performance is comparable to popular high-performance general-purpose hypervisors. We verify memory integrity of XMHF following the DRIVE methodology. Most of the DRIVE verification conditions are discharged using the software model checker CBMC [22]. Out of the 6018 lines of code that comprise the XMHF core, CBMC automatically verifies 5208 lines in about 80 seconds using less than 2GB of RAM. We manually audit the remaining 422 lines of C code and 388 lines of assembly language code which we anticipate to remain mostly unchanged as development proceeds. The manual audits include constructs that CBMC cannot verify, including loops that iterate over entire page tables, platform hardware initialization and interaction, and concurrent threads coordinating multiple CPUs that are challenging for current model-checkers.

We list the contributions below, which also serves as a roadmap to the paper:

- We present the DRIVE methodology for designing, developing, and verifying hypervisor *memory integrity* (§III).
- We design and implement XMHF, a hypervisor framework based on DRIVE which supports modular development of future hypapps (§IV).
- We verify the memory integrity of the XMHF runtime implementation using DRIVE, and show how to discharge the DRIVE verification conditions on XMHF using the software model checker CBMC [22] (§V).
- We carry out a comprehensive performance evaluation of XMHF (§VI).

II. Goals, Background and Attacker Model

A. Modular Development and Verification

Our overarching goal is to create a hypervisor framework that promotes *development* of custom hypapps, while simultaneously allowing *verification* of security properties. We focus on verifying memory-integrity – a fundamental security property and a major component of the tamperproofness of any hypapp. This enables the development of hypapps without having to worry about the low-level infrastructure grunge or the hypervisor's memory integrity.

We strive for a minimal TCB hypervisor design that enables automatic verification of its implementation. Accordingly, we propose a *rich* single-guest execution model (§IV-A). Thus, XMHF supports only a single-guest that directly accesses and manages platform devices after initialization. XMHF consists of a core and small supporting libraries. These are extended, and leveraged, by each hypapp to implement its functionality. Our specific design goals are:

1) Modular Core and Modular Extensibility: The XMHF core is built in a *modular* way. It handles a set of events from the guest (e.g., hypercall, nested page-faults) using *distinct* event handlers. Each event is handled in a sequential manner, either directly by the XMHF core or handed over to a hypapp handler, which then performs the desired functionality by leveraging a set of APIs exposed by XMHF (§IV-B1). This *modular extensibility* allows a hypapp to extend XMHF to offer custom features and desired properties.

2) Verifiability: The rich single-guest model results in reduced hypervisor complexity (since all devices are directly controlled by the guest) and consequently TCB. Further, XMHF's *modular core* design allows independent automated analysis of the event handlers, which constitute the runtime attack surface. In particular, it enables a software model checker to ignore irrelevant code (e.g., via slicing) when verifying a particular property (§V).

3) Performance: The rich single-guest model promises high guest performance as all performance-critical system devices and device interrupts are directly handled by the guest without the intervention of XMHF (§VI-B).

B. Hardware Virtualization Primitives

We focus on the following hardware virtualization primitives offered by the system platform. These primitives are supported by current x86 platforms [23], [24], and are also making their way on embedded ARM architectures [25].

- The CPU executes in two modes, each with a separate address space: (a) *host-mode* (or privileged mode) – where the hypervisor executes, and (b) *guest-mode* (or unprivileged mode) – where the guests execute.
- At system boot time, the hypervisor is able to execute a designated piece of code in complete isolation.
- At system runtime, the hardware provides mechanisms to ensure that all accesses to system memory are subjected to an access control mechanism.
- The execution state of the guest is maintained in a data structure that is inaccessible and/or access controlled from unprivileged mode.
- The hypervisor is able to associate *intercept handlers* with certain events caused by a guest (e.g., instructions, I/O operations, exceptions and interrupts). The hardware ensures that upon the occurrence of an event e, the following sequence of actions occur: (1) the execution state of the guest is saved, (2) execution is switched to host mode, (3) the intercept handler for e is executed, (4) execution is switched back to guest mode, and (5) the execution state of the guest is restored and guest execution is resumed.

C. Attacker Model

We consider attackers that do not have physical access to the CPU, memory and the chipset (our hardware TCB). Other system devices and the guest constitute the attacker. This is a reasonable model since a majority of today's attacks are mounted by malicious software or system devices. An attacker can attempt to access memory either (i) during hypervisor initialization; or (ii) from within the guest and by using system devices; or (iii) via hypervisor intercept handlers triggered by the guest.

D. System Assumptions

We assume that the our hardware TCB provides the correct functionality and that the hypervisor has control flow integrity [26], i.e., the control flow during the execution of the hypervisor respects its source code. Ensuring CFI for systems software is an orthogonal challenge to verifying memory integrity. As future work, we plan to reinforce XMHF with CFI and verify its correctness. We also assume that a hypapp built on top of XMHF uses the prescribed XMHF core APIs (e.g., changing guest memory protections and performing chipset I/O) and that it does not write to arbitrary code and data. In fact, these are the only assumptions required of any new hypapp to ensure the memory integrity property of XMHF. We plan to explore modular verification of the XMHF core composed with hypapps as future work.

III. The DRIVE Methodology

We model the virtualized system as a tuple $V = (H, G, D, \mathcal{M})$, where H is the hypervisor, G represents the guest, D represents devices, and $\mathcal{M}$ is the hypervisor memory containing both hypervisor code and data. Both G and D are attacker controlled. We omit the guest memory, which is separate from $\mathcal{M}$ and irrelevant to memory integrity, from the model. DRIVE consists of a set of properties about H, system invariants, and a proof that if H satisfies those properties then the invariants hold on all executions of V. This, in turn, implies the memory integrity of H in V.

A. Hypervisor Properties Required by DRIVE

DRIVE identifies the following six properties that restrict the hypervisor design and implementation, response to the attacker's actions, and writes to memory.

Modularity (MOD). Upon hypervisor initialization, control is transferred to a function $init()$. When an intercept is triggered, the hardware transfers control to one of the intercept handlers $ih_1(), \ldots, ih_k()$.

Atomicity (ATOM). This property ensures the atomicity of initialization and intercept handling on the CPU(s). It consists of two sub-properties: ATOM$_{init}$ – at the start of V's execution, $init(\,)$ runs completely in a single-threaded environment before any other code executes; ATOM$_{ih}$ – the intercept handlers $ih_1(), \ldots, ih_k()$ always execute in a single-threaded environment.

Memory Access Control Protection (MPROT). H uses a memory access control mechanism *MacM*. All *MacM* related state is stored in $\mathcal{M}$. *MacM* consists of two parts: (1) *MacM*$_G$ – for the guest, and (2) *MacM*$_D$ – for the devices.

Correct Initialization (INIT). After H's initialization, *MacM* protects $\mathcal{M}$ from the guest and devices. The intercept entry points into H points to the correct intercept handler.

Proper Mediation (MED). *MacM* is active whenever attacker-controlled programs execute. This implies: (1) before control is transfered to the guest (G), the CPU is set to execute in guest mode to ensure that *MacM*$_G$ is active, and (2) *MacM*$_D$ is always active.

Safe State Updates (SAFEUPD). All updates to system state including $\mathcal{M}$ and control structures of the hardware TCB (e.g., guest execution state and chipset I/O), by an intercept handler: (1) preserve the protection of $\mathcal{M}$ by *MacM* in guest mode and for all devices; and (2) do not modify the intercept entry point into H, and (3) do not modify H's code.

B. System Invariants

We define two system invariants for V that imply H's memory integrity. We say that V preserves an invariant φ, if when $init()$ finishes, φ holds; and at all intermediate points during the execution of V, φ holds. The memory invariants, denoted $\varphi_{\mathcal{M}}$, require that $\mathcal{M}$ is properly protected and that both the entry point to H and the code stored in the entry point have not been modified. Invariant φ_{Med} requires that DMA protection has not been disabled.

$$\begin{aligned} \varphi_{\mathcal{M}} &= \mathcal{M} \text{ is designated read-only in } \mathit{MacM} \text{ and} \\ &\quad\ \text{intercept } i \text{ jumps to the starting address of } ih_i(\,). \\ \varphi_{Med} &= \mathit{MacM}_D \text{ is always active.} \end{aligned}$$

Informally, ensuring that these invariants hold on all executions of V requires both H and G to preserve these invariants. The properties in Section III-A entail that H preserves the invariants. The hardware and the protections set up prior to the execution of the guest ensures that G cannot violate the invariants as well.

Based on the system invariants and DRIVE properties, we extract a sequentialized execution model for V, which makes automated verification of DRIVE properties feasible. As we discuss in Section V, we use a software model checker CBMC to verify properties of the C implementation of XMHF. CBMC assumes sequential execution, and therefore, the sequentialized execution model makes sure that using CBMC is appropriate. Properties MOD and ATOM allow V's execution to be sequentialized assuming that the entry point from G to H remain unchanged. The first step after system power-on is initialization. Subsequently, the system executes either: (1) G (e.g., guest OS) in unprivileged mode; or (2) D (e.g., network and graphics card) and is able to perform direct memory accesses (DMA); or (3) H (e.g., intercept handlers triggered by G) in privileged mode.

Given two sequential programs f and g, $f + g$ denotes the sequential program that non-deterministically executes either f or g (but not both), and $f \mid g$ denotes the parallel composition of f and g. Both $+$ and $\mid$ are commutative and associative. We write $f(*)$ to denote the execution of function f given an arbitrary input. The sequentialized executions of V, denoted $\mathit{Seq}(V)$ is defined formally as:

$$\begin{aligned} \mathit{Seq}(V) = {} & \mathit{init}(*); \\ & \mathit{while}(\mathsf{true})\ \{(G + ih_1(*) + \cdots + ih_k(*)) \mid D\} \end{aligned}$$

C. Proof of Memory Integrity

The key part of the proof is Lemma 1 stating that the hypervisor properties ensure the invariants $\varphi_{\mathcal{M}}$ and φ_{Med} hold at all times on all execution traces of V. In other work, we have formally modeled the program logic of V and verified Lemma 1 using a novel logic [27]. Briefly, the proof is by induction over the length of the execution trace.

Lemma 1. *If H satisfies* MOD, ATOM, MPROT, MED, INIT *and* SAFEUPD, *then $\varphi_{\mathcal{M}}$ and φ_{Med} are invariants of all executions of V.*

Theorem 2. *If H satisfies* MOD, ATOM, MPROT, MED, INIT *and* SAFEUPD, *then in all executions of V, any write to $\mathcal{M}$ after initialization is within $ih_i(\,)$.*

Proof (sketch): Given any write to $\mathcal{M}$, using Lemma 1 and the property of the hardware, we know that write must occur in host mode. Thus, it must be the case that the guest has exited to enter H. Using MOD and ATOM and that the entry point to H and the code of H has not been modified, we know that the write must have been called from one of the intercept handlers $ih_i(\,)$. □

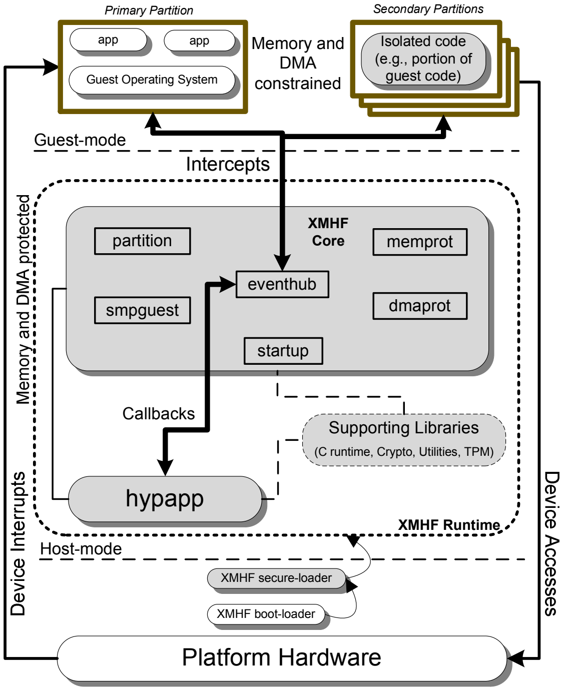

Figure 1. XMHF *rich* single-guest architecture. XMHF consists of the XMHF core and small supporting libraries that sit directly on top of the platform hardware. A hypapp extends the XMHF core to implement the desired (security) functionality. XMHF allows the guest direct access to all performance-critical system devices and device interrupts resulting in reduced hypervisor complexity, consequently Trusted Computing Base (TCB), as well as high guest performance. Shaded portions represent code isolated from the rich guest.

IV. XMHF DESIGN AND IMPLEMENTATION

We highlight the design and implementation decisions that help make XMHF minimalistic, enable verification of DRIVE properties on XMHF's C implementation, and make automated re-verification in the process of hypapp development possible. We first discuss the *rich* single-guest execution model of XMHF, show how it enables us to achieve our design goals (§II), and provide details of its implementation. We then show how XMHF's design and implementation achieve the properties mandated by DRIVE to ensure memory integrity. The high-level design principles behind XMHF are platform independent. The XMHF implementation currently supports both Intel and AMD x86 hardware virtualized platforms, and unmodified multi-processor Windows (2003 and XP) and Linux as guests. However, XMHF design principles apply to other architectures, such as ARM, as well.

A. Rich Single-guest Execution Model

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 a guest OS. The guest runs on another (unprivileged) level above the hypervisor. The bare-metal design allows for a small-TCB and high performance hypervisor code base. Recall that XMHF consists of the XMHF core and small supporting libraries that sit directly on top of the platform hardware. A hypapp extends the XMHF core and leverages the basic hypervisor and platform functionality provided by the core to implement the desired (security) functionality (Figure 1). XMHF supports only a single-guest and allows the guest to directly access and manage platform devices. The single-guest model allows (cf. Cloudvisor [5] and Turtles [28]) its guest to be another (more traditional) hypervisor running multiple guest OSes[1].

1) Achieving XMHF *Design Goals:* We now describe how the rich single-guest model enables us to achieve XMHF's design goals previously presented in §II.

Modular Core and Modular Extensibility: In the rich single-guest execution model, the hypervisor interacts with the guest via a well-defined hardware platform interface. In XMHF, this interface is handled by the XMHF core or hypapp handlers. The XMHF core and supporting libraries expose a small set of APIs that allow a hypapp to extend the XMHF core to offer custom features and security guarantees.

Verifiability: Since all devices are controlled directly by the guest, XMHF does not have to deal with per-device idiosyncrasies that arise from devices that are not completely standards-compliant. In addition, XMHF does not need to perform hardware multiplexing, an inherently complex mechanism that can lead to security issues [30], [31]. This results in a small and simple hypervisor code-base. Further, the system devices (including interrupt controllers) are directly in control of the guest. Therefore, all (device) interrupts are configured and handled by the guest without the intervention of XMHF. This allows XMHF to be designed for sequential execution (i.e., no interrupts within the hypervisor) while at the same time allowing the guest to use multiple CPUs and be concurrent. The sequentialization together with the small and simple hypervisor code-base enables us to discharge DRIVE verification conditions on the XMHF code-base automatically using a software model checker (§V).

Performance: Since all (device) interrupts are configured and handled by the guest without the intervention of XMHF, guest performance overhead is minimal (the guest still incurs hardware memory/DMA protection overhead) and comparable to popular high-performance general-purpose hypervisors (§VI-B).

2) Implementation Features: We discuss the salient implementation features of the XMHF rich single-guest execution model below.

Prevent access to critical system devices: Critical system devices – such as the DMA protection hardware and the system memory controller – expose their interfaces through either legacy or memory-mapped I/O. For example, Intel and AMD x86 platforms expose the DMA protection hardware through the Advanced Configuration and Power Management Interface (ACPI) and the Peripheral Component Interconnect (PCI) subsystems, respectively. With the rich single-guest model, the guest could perform direct I/O to these devices, effectively compromising the memory and DMA protections. XMHF marks the ACPI and PCI configuration space of critical system devices as *not-present* using the Hardware Page Tables (HPT) (see § IV-B3), and makes the memory-mapped I/O space of these devices inaccessible to the guest. A well-behaved guest OS should never attempt to access these regions.

Guest memory reporting: A native OS during its bootup uses the BIOS (`INT 15h E820` interface) to determine the amount of physical memory in the system. However, with XMHF loaded, there must be a mechanism to report a reduced memory map excluding the hypervisor memory regions to the guest. If not, the guest at some point during its initialization will end up accessing the protected hypervisor memory areas, which is difficult to recover from gracefully. Currently, this causes XMHF to halt. XMHF leverages HPTs to report a custom system memory map to the guest. During initialization XMHF replaces the original `INT 15` BIOS interface handler with a hypercall instruction. The XMHF hypercall handler then presents a custom memory map with the hypervisor memory region marked as *reserved* and resumes the guest.

B. Ensuring Memory Integrity

To achieve DRIVE properties, XMHF relies on platform hardware support, which includes hardware virtualization, two-level Hardware Page Tables (HPT), DMA protection, and dynamic root of trust (DRT) support. These capabilities are found on recent Intel and AMD x86 platforms. Similar capabilities are also forthcoming in ARM processor platforms [25]. While this breaks backward compatibility with older hardware, it allows XMHF's design to be much smaller and cleaner while achieving the DRIVE properties to ensure memory integrity.

1) Ensuring MOD*:* The XMHF core and a hypapp interact with a guest environment via an event-based interface. Unlike regular application interfaces, this event-based interface is supported by the underlying hardware and is well-defined. The XMHF core only handles a small subset of this interface and allows a hypapp to configure and handle only the required events. This reduces the interface surface and avoids unnecessary guest event traps at runtime. Nevertheless, the

[1] This requires emulation of hardware virtualization support, which is feasible in around 1000 lines of additional code as evidenced by KVM [29].

event-based interface is versatile enough to enable development of a variety of applications with interesting security properties and functionality [1]–[11], [13]–[16], [32].

XMHF leverages CPU support for hardware virtualization to capture and handle events caused by a guest operating environment. For example, recent x86 and embedded ARM hardware virtualized platforms define a set of *intercepts* that transfer control to the hypervisor upon detecting certain guest conditions [23]–[25]. The XMHF core gets control for all intercepted guest events and in turn invokes the corresponding XMHF/hypapp callback to handle the event (Figure 1). 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 hypapps to easily extend core XMHF functionality to realize desired functionality in the context of a particular guest.

Both Intel and AMD x86 platforms transfer control to a single designated entry point within the hypervisor upon a guest intercept. The core's *eventhub* component is the top-level intercept entry point in XMHF. For each intercepted class of event the *eventhub* component invokes a distinct hypapp callback with the associated parameters in the context of the CPU on which the intercept was triggered.

2) Ensuring ATOM*:* For the initialization $init()$, XMHF leverages DRT to ensure its execution atomicity (ATOM$_{init}$). A 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. XMHF's launch process consists of a XMHF *boot-loader* that establishes a DRT and loads the XMHF *secure-loader* in a memory constrained hardware protected environment (Figure 1).

The XMHF *boot-loader* uses the `GETSEC[SENTER]` and `SKINIT` CPU instructions on Intel and AMD x86 platforms respectively, to create a DRT and bootstrap the XMHF *secure-loader* in a memory-protected single-threaded environment. The XMHF *secure-loader* in turn sets up the initial memory paging and DMA protection and transfers control to the XMHF core *startup* component which performs the runtime initialization.

The rich single-guest execution model allows XMHF to be designed for sequential execution (i.e., without any interrupts within the hypervisor). However, on multicore platforms there can still be concurrent execution within the hypervisor during intercept handling. Thus, to ensure ATOM$_{ih}$, XMHF uses a technique called *CPU-quiescing*. Using CPU-quiescing, the moment an intercept is triggered on a specific CPU, XMHF stalls the remaining CPUs. Once the intercept has been handled, the stalled CPUs are resumed and control is transferred back to the guest. The quiescing latency is low enough so as not to break any delay-sensitive device I/O (see §VI-B2).

XMHF uses the Non-Maskable Interrupt (NMI) for *CPU-quiescing*. Specifically, when an intercept is triggered on a CPU C, XMHF acquires a global intercept lock and sends an NMI to all other CPUs (excluding C itself). Since the NMI cannot be masked, this causes the target CPUs to receive an NMI. The NMI handler is invoked, which is an idle spin-lock loop that stalls the CPU on which it runs. Once the intercept has been handled on C, XMHF signals the spin-lock which causes the other CPUs to resume.

Note that DRT automatically disables NMI generation, so the XMHF *secure-loader* and runtime initialization are guaranteed to be single-threaded. The XMHF core then sets up the NMI handler as described above. At runtime, a NMI handler execution on a given CPU is also guaranteed to be atomic. The CPU initiating the quiescing will wait for all the other CPU NMI handlers to enter the idle spin-lock loop before proceeding to execute an intercept handler.

3) Ensuring MPROT*:* XMHF uses HPTs for efficient guest memory access control. 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 indicating a violation.

XMHF leverages hardware DMA protections to protect its memory regions from direct access by devices. The DMA protection is part of the hardware platform (e.g., chipset) and is specified using a DMA table within XMHF. In particular, the hardware ensures that all memory accesses by system devices are translated using the permissions specified within the DMA table. The hardware disallows DMA if the access requested violates the permissions stored in the DMA table.

XMHF uses the Extended Page Tables (EPT) and Nested Page Tables (NPT) on Intel and AMD x86 platforms respectively for guest memory access control. On Intel platforms, XMHF uses the VT-d (Virtualization Technology for Directed I/O) support to provide DMA protection. The VT-d page tables contain mappings and protections for the DMA address space as seen by system devices. On AMD platforms, XMHF relies on the Device Exclusion Vector (DEV) for DMA protection. DEV's bitmap structure allows DMA protection to be set for a given memory address range.

4) Ensuring INIT*:* During initialization XMHF sets up the HPTs and DMA table permissions so that memory addresses corresponding to the hypervisor memory regions are marked *read-only* and therefore cannot be modified by either the guest or system devices.

The XMHF core *memprot* and *dmaprot* components set up the EPT/NPTs and the VT-d/DEV DMA protection permissions on Intel and AMD x86 platforms respectively.

5) Ensuring MED*:* During initialization, XMHF activates the platform hardware DMA protection mechanism that enforces DMA access control for hypervisor memory accesses by system devices. More concretely, the XMHF core *dmaprot* component activates the VT-d and DEV DMA protection mechanisms on Intel and AMD x86 platforms respectively, to prevent system devices from accessing memory regions belonging to XMHF.

Access control protections for guest memory accesses are described by the HPTs and enforced by the CPU when operating within guest-mode. XMHF uses *partitions*[2] to contain guest code and data. A partition is essentially a bare-bones CPU hardware-backed execution container that enforces system memory isolation for the guest or a portion of it based on HPTs that are initialized by the hypervisor. XMHF creates a primary partition in order to run the guest operating environment. XMHF can also instantiate secondary partitions on demand when requested by a hypapp. These secondary partitions are capable of running specified code within a low-complexity isolated environment, which is explicitly designed without support for scheduling or device interaction (Figure 1). This is useful when a hypapp wishes to implement desired security properties at a fine granularity, e.g., portions of an untrusted application within the operating environment (e.g., TrustVisor [7] and Alibi [1]).

The XMHF core *partition* component uses the `VMLAUNCH/VMRESUME` and `VMRUN` CPU instructions on Intel and AMD x86 platforms respectively to instantiate partitions. The following paragraphs describe how XMHF supports multi-processor guests while ensuring hypervisor memory protection.

On x86 platforms, only one CPU – called the boot-strap processor (BSP) – is enabled when the system starts. The other CPUs remain in halted state until activated by software running on the BSP. During its initialization, XMHF activates the remaining CPUs and switches all the CPUs (including the BSP) to host-mode. Next, XMHF sets up the HPTs on all the cores and switches the BSP to guest-mode to start the guest; the remaining CPUs idle in host-mode within XMHF. Finally, the XMHF core *smpguest* component uses a combination of HPTs and intercept handling (described below) to ensure that the remaining cores are switched to guest-mode before they execute guest code. This ensures that HPT access control is always enabled for all CPU cores.

A native multicore capable OS, on the x86 platform, uses the CPU Local Advanced Programmable Interrupt Controller (LAPIC) for multicore CPU initialization [23], [24]. More specifically, the LAPIC Interrupt Control Register (ICR) is used to deliver the startup sequence to a target core. On x86 platforms, the LAPIC is accessed via memory-mapped I/O encompassing a single physical memory page. XMHF leverages Hardware Page Tables (HPTs) to trap and intercept accesses to the LAPIC memory page by the guest OS.

Subsequently, any writes to the LAPIC ICR by the guest causes the hardware to trigger a HPT violation 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 startup command is written to the ICR, XMHF voids the instruction and instead runs the target guest code on that core in guest-mode.

6) Ensuring SAFEUPD*:* XMHF requires both the XMHF core and the hypapp to use a set of well-defined interfaces provided by the core to perform all changes to the HPTs. These interfaces upon completion, ensure that permissions for the hypervisor memory regions remain unchanged. In the current XMHF implementation, a single XMHF core API function `setprot` provided by the core *memprot* component allows manipulating guest memory protections via the HPTs.

V. XMHF VERIFICATION

In this section we present our verification efforts on the XMHF implementation. We discuss which DRIVE properties are manually audited and why they are likely to remain valid during XMHF development. We also show how most of the verifications of DRIVE properties are reduced to inserting assertions in XMHF's source code, which is then checked automatically using CBMC [22]. As our focus is on verifying the memory integrity of the XMHF core, we use a simple hypapp for verification purposes. The hypapp implements a single hypercall interface to manipulate guest memory protections via Hardware Page Tables (HPT).

A. Overview

The XMHF verification process is largely automated. We manually audit 422 lines of C code and 388 lines of assembly language code. The manual auditing applies to functions in XMHF core that are unlikely to change as development proceeds. The automatic verfication of 5208 lines of C code uses CBMC.

In addition to the system assumptions presented in §II, the soundness of our verification results depends on two additional assumptions: (i) CBMC is sound. i.e., if CBMC reports that XMHF passes an assertion, then all executions of XMHF satisfy that assertion; and (ii) the XMHF core interface – determined by the available types of hardware virtualization intercepts – is complete, i.e., XMHF handles all possible intercepts from guests.

*B. Verifying Modularity (*MOD*)*

We verify MOD by engineering the source code of XMHF to ensure that the implementations of $init()$ and

[2] Such CPU execution containers are often called hardware virtual machines in current parlance. However, this is a misnomer in our case since, technically, a virtual machine presents to the guest a virtualized view of the system devices in addition to enforcing memory isolation.

$ih_1(), \dots, ih_k()$ are modular (recall §III-A). The XMHF core *startup* component implements the $init()$ function in XMHF. It first performs required platform initialization, initializes memory such that MProt holds, then starts the guest in guest-mode. The XMHF core *eventhub* component implements $ih_1(), \dots, ih_k()$ in XMHF. More specifically, the *eventhub* component consists of a single top-level intercept handler function which is called whenever any guest intercept is triggered. We refer to this function as $ihub()$. The arguments of $ihub()$ indicate the actual intercept that was triggered. Based on the value of these arguments, $ihub()$ executes an appropriate sub-handler.

*C. Verifying Atomicity (*Atom*)*

We rely on the hardware semantics of Dynamic Root-of-Trust (DRT) (§IV-B2) to discharge Atom$_{init}$. There are preliminary verification results of the correctness of DRT at the design-level [33], which forms the basis of our assumptions on DRT's semantics.

We check Atom$_{ih}$ by manually auditing the functions implementing CPU-quiescing (§IV-B2). More specifically we manually audit three C functions which are responsible for stalling and resuming the CPUs and for handling the NMI used for CPU quiescing, to ensure proper intercept serialization. While these checks are done manually, we believe that it is acceptable for several reasons. First, the functions total to only 60 lines of C code. They are largely self-contained (no dependent functions and only four global variables) and invoked as wrappers at the beginning and end of the intercept handlers. Therefore, we only need to perform manual re-auditing if any of the CPU-quiescing functions themselves change. Given the simple design and functionality of quiescing, and based on our development experience so far, we anticipate the quiescing functions to remain mostly unchanged as development proceeds.

D. Verifying MProt

MProt is always preserved by XMHF since the HPTs and DMA protection data structures are statically allocated. We verify MProt automatically by employing a build script that inspects relevant symbol locations in the object and executable files produced during the build process to ensure that the DMA protection and HPT data structures reside within the correct data section in hypervisor memory $\mathcal{M}$.

E. Verifying Init

Init is checked by a combination of manual audits and automatic verification on the XMHF source to ensure that before XMHF's $init()$ function completes, the following are true: DMA table and HPTs are correctly initialized so that memory addresses corresponding to the hypervisor memory regions cannot be changed by either the guest or system devices; and the intercept entry point in XMHF points to $ihub()$.

```
//start a partition by switching to guest-mode
//cpu = CPU where the partition is started
void xmhf_partition_start(int cpu)
{
  ...
#ifdef VERIFY
   assert( cpu_HPT_enabled );
   assert( cpu_HPT_base == HPT_base );
   assert( cpu_intercept_handler == ihub );
#endif
  //switch to guest-mode
}
```

Figure 2. Outline of `xmhf_partition_start`, the function used to execute a target `cpu` in guest-mode. `cpu_HPT_enabled` and `cpu_HPT_base` enforce hardware page table (HPT) protections. `cpu_intercept_handler` is where the CPU transfers control to when an intercept is triggered in guest-mode. `HPT_base` and `ihub` are the XMHF initialized HPTs and the intercept handler hub respectively. These assertions allow automatic verification of DRIVE properties Init and Med in XMHF using a model checker.

The manual audits involve 311 lines of C code and 338 lines of assembly language code which include platform hardware initialization, loops including runtime paging and DMA table and HPT setup, and concurrency in the form of multicore initialization within XMHF. Given the stable hardware platform and multicore initialization logic as well as paging, DMA and HPT data structure initialization requirements, we postulate that the manually audited code will remain mostly unchanged as development proceeds, ensuring minimal manual re-auditing effort.

1) DMA table and HPT initialization: The XMHF *secure-loader* and the XMHF core *startup* component set up the DMA table to prevent system devices from accessing XMHF memory regions. We manually audit the C functions responsible for setting up the DMA table to verify that the functions assign entries in the DMA table such that all addresses in $\mathcal{M}$ are designated *read-only* by devices.

Before the XMHF $init()$ function transfers control to the guest, it calls a C function to setup the HPTs for the guest. We manually audit this C function to verify that the function assigns each entry in the HPTs such that all addresses in $\mathcal{M}$ are designated read-only by the guest.

2) Intercept entry point: The XMHF $init()$ function finally invokes the `xmhf_partition_start` function to start the guest in guest-mode. We insert an assertion in `xmhf_partition_start` and use CBMC to automatically verify the assertion. The inserted assertion in `xmhf_partition_start` checks that the CPU is setup to transfer control to $ihub()$ on an intercept (Figure 2).

F. Verifying Med

Med is verified by ensuring the following: (1) XMHF's $init()$ function in the end sets the CPU to execute in guest-mode with the appropriate hardware memory protections in place. In particular, in a multicore system, every CPU that is initialized is set to execute in guest-mode. (2) DMA protection is always active during XMHF runtime. We insert assertions in XMHF source code and use CBMC to automatically verify these assertions.

1) Guest-mode execution: XMHF initializes memory protection for a CPU and finally uses the `xmhf_partition_start` function to execute a target CPU in guest-mode. Therefore, we verify that CPU transitions to the correct guest-mode by model-checking the validity of assertions inserted in the `xmhf_partition_start` function (Figure 2). The assertions check that appropriate fields in the MMU data structures are set to point to the correct HPTs and that HPT protections are in effect before using the CPU instruction that performs the switch to guest-mode.

On a multicore platform, XMHF uses a combination of the nested page fault (*ih*_`npf`) and the single-step (*ih*_`db`) intercepts to ensure that each CPU that is initialized is set to execute in guest-mode (via return from *ihub*()) before it executes any attacker code (recall §IV-B5). More specifically, XMHF maps the LAPIC to a page in $\mathcal{M}$ during initialization. Subsequently, assuming SafeUpd holds, ensuring Med on multicore systems reduces to verifying that: (i) *ih*_`npf`() disables guest interrupts and sets the guest trap-flag on access to the LAPIC memory-mapped I/O page, and (ii) *ih*_`db`() prevents a direct write to the LAPIC Interrupt Control Register (ICR) upon detecting a startup command and instead runs the target guest code on C in *guest-mode*.

We check this property by a combination of manual audits and automatic verification by model-checking the validity of assertions inserted in *ihub*(), *ih*_`npf` and *ih*_`db` (Figure 3). The assertions check that the appropriate core is switched to guest-mode if the single-step intercept is triggered. The manual audits comprise of 51 lines of C code which correspond to the LAPIC page mapping setup and handling within *init*(), *ih*_`npf` and *ih*_`db`.

2) DMA protection: The XMHF *secure-loader* is started via a DRT operation which ensures that the *secure-loader* memory is automatically DMA-protected by the hardware (§IV-B2). The XMHF *secure-loader* activates DMA protection for the XMHF runtime before transferring control to the XMHF core *startup* component which, during initialization, re-activates DMA protection before transferring control to the guest. The DMA protection activation is done by setting the appropriate bits in the DMA protection hardware registers to enforce DMA protection. We verify this by model-checking the validity of a properly inserted assertion in XMHF, as shown in Figure 4. The inserted assertion checks that the relevant (enable) bit is set in the DMA protection hardware register value before writing to the register to enable DMA protection.

G. Verifying SafeUpd

Under the assumption that the XMHF core and the hypapp only modify guest memory protections through the core API function `setprot`, and that neither the XMHF core or the hypapp modify their own code regions and the entry point to the hypervisor, verification of SafeUpd is reduced to checking that when `setprot` completes, all hypervisor memory addresses are still protected. We additionally know that after XMHF's *init*() function completes, the hypervisor memory is protected (§V-E), so we further reduce the verification obligation to checking that `setprot` does not alter the protection bits of hypervisor memory regions

```
//top-level intercept handler
//cpu = CPU where intercept triggered
//x = triggered intercept
void ihub(int cpu, int x){
  //...main body of ihub
#ifdef VERIFY
  assert( cpu_HPT_enabled );
  assert( cpu_HPT_base == HPT_base );
  assert( cpu_intercept_handler == ihub );
#endif
}

//nested page fault handler
void ih_npfe(){
#ifdef VERIFY
  int pre_npfe = NPFELAPIC_TRIGGERED();
#endif
  //...main body of ih_npfe
#ifdef VERIFY
  assert (!pre_npfe || GUEST_TRAPPING(cpu));
#endif
}

//single-step exception handler
void ih_db(){
#ifdef VERIFY
  int pre_dbe = LAPIC_ICRWRITE();
#endif
  //...main body of ih_db
#ifdef VERIFY
  assert (!pre_dbe || CORE_PROTECTED(cpu));
#endif
}
```

Figure 3. Outline of *ihub*(), the top-level intercept handler function, and *ih*_`npf` and *ih*_`db`, the nested page fault and single-step handlers. `NPFELAPIC_TRIGGERED()` = true iff the `npf` intercept was triggered in response to the guest accessing the LAPIC memory-mapped I/O page. `GUEST_TRAPPING(cpu)` = true iff the CPU identified by `cpu` has interrupts disabled and is set to generate a `db` intercept. `LAPIC_ICRWRITE()` = true iff a write was performed to the LAPIC ICR. `CORE_PROTECTED(cpu)` = true iff the LAPIC ICR write was disallowed upon detecting a startup command. `cpu_HPT_enabled` and `cpu_HPT_base` are the MMU fields that enforce hardware page tables (HPT). `cpu_intercept_handler` is the field that the CPU transfers control to when an intercept is triggered in guest-mode. `HPT_base` and `ihub` are the XMHF initialized HPTs and the intercept handler hub respectively. These macros and assertions allow a model checker to automatically verify DRIVE Med in XMHF.

Similar to before, we use CBMC to automatically verify `setprot` does not alter the protection bit of the hypervisor memory by inserting an assertion preceding every write operation to the HPTs, checking that the address being written to does not belong to the hypervisor. More concretely, the hypervisor memory is maintained in a contiguous set of addresses beginning at `HVLO` and ending at `HVHI`. Therefore, every statement that potentially modifies the permission of

```
//activate DMA protection
  ...
#ifdef VERIFY
  assert( controlreg_value & DMAP_ENABLE );
#endif
  DMAPwrite(controlreg, controlreg_value);
  ...
```

Figure 4. Outline of DMA protection activation in XMHF. `controlreg` and `controlreg_value` are the DMA protection hardware control register and register value respectively. `DMAP_ENABLE` enables the DMA protection and `DMAPwrite` writes a value to a given DMA protection hardware register. The assertion allows a model checker to automatically verify DMA protection activation and consequently DRIVE MED in XMHF.

```
//set permission of address a to p
void xmhf_memprot_setprot(int a,int p)
{
  ...
  //the following assertion precedes every
  //statement that sets permission of
  //address a to p
#ifdef VERIFY
  assert (a < HVLO || a > HVHI);
#endif
  ...
}
```

Figure 5. Outline of the XMHF core API function `setprot`, which is used to modify guest memory protections via the Hardware Page Tables (HPT). The assertion allows a model-checker to verify DRIVE SAFEUPD.

a memory address a is preceded by an assertion that checks that $a < \texttt{HVLO} \vee a > \texttt{HVHI}$, as shown in Figure 5.

H. Discussion

1) Modular Development and Verification: XMHF's verification is intended to be used as part of the XMHF build process automatically. Developers of hypapps are not required to deal with the verification directly. This is similar to other approaches such as the SDV tool [34] which packages the device driver verifier as part of the driver verifier kit for Windows. Developers, however, must adhere to the prescribed XMHF core APIs when changing guest memory protections or accessing the hardware TCB control structures (e.g., performing chipset I/O or accessing hardware virtual machine control structures). Developers must also ensure that hypapp code does not perform writes to code or write to arbitrary data that is not part of the hypapp.

Note that the assertions and verification statements inserted in the XMHF code are for use by CBMC only. Once CBMC reports a successful verification, these statements and assertions are proven unnecessary, and can therefore be removed in the production version of XMHF, so that they do not hinder performance.

2) Manual Audits: The manual audits described in the previous sections include constructs that CBMC cannot verify, including loops that iterate over entire page tables (e.g., runtime paging, DMA table and HPTs), platform hardware initialization and interaction (e.g., CPU, LAPIC, BIOS and PCI) and concurrency (e.g., multicore initialization within XMHF and multicore guest setup). These are verification challenges that continue to garner attention from the research community. For example, a number of other tools (see [35] for a list) are being developed for verifying concurrent C programs. There are design-level verification techniques [36], [37] that could be employed to address the scalability problem (e.g. loops) with current model-checkers, for hypervisor designs. We plan to explore their applicability in the future.

VI. 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. We also 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. Finally, we discuss our verification results.

A. XMHF *TCB and Case Studies with hypapps*

XMHF's TCB consists of the XMHF core, the hypapp and supporting libraries used by the hypapp. The XMHF supporting libraries (totaling around 8K lines of C code) currently include a tiny C runtime library, a small library of cryptographic functions, a library with optional utility functions such as hardware page table abstractions and command line parsing functions, and a small library to perform useful TPM operations. From a hypapp's perspective, the minimum TCB exposed by XMHF comprises the XMHF core which consists of 6018 SLoC.

We demonstrate the utility of XMHF as a common framework for developing hypapps by porting several recent open-source research efforts in the hypervisor space to XMHF. Figure 6 shows the SLoC metrics and platform support for each hypapp before and after the port to XMHF. TrustVisor [7] and Lockdown [2] are fully functional, and their code sizes are precise. The development of HyperDbg [8], XTRec [6] and SecVisor [10] is sufficiently advanced to enable estimation of their final sizes via manual inspection of their existing sources and differentiation between the hypervisor core and hypapp-specific logic. Figure 6 shows that the XMHF core forms 48% of a hypapp's TCB, on average. This supports our hypothesis that these hypervisors share a common hypervisor core that is re-used or engineered from scratch with every new application. Also, using XMHF endows the hypapps with support for x86 muticore platforms from both Intel and AMD for free.

B. Performance Measurements

We measure XMHF's runtime performance using two metrics: 1) guest overhead imposed solely by the framework

	Original			On XMHF					
hypapp	SLoC	Arch. Support	Multicore Support	XMHF core SLoC	hypapp + libs. SLoC	Total SLoC	% XMHF core	Arch. Support	Multicore Support
TrustVisor	6481	x86 AMD	No	6018	9138	15156	40%	x86 AMD, Intel	Yes
Lockdown	~10000	x86 AMD	No	6018	9391	15409	40%	x86 AMD, Intel	Yes
XTRec*	2195	x86 AMD	No	6018	3500*	9500*	63%*	x86 AMD, Intel	Yes
SecVisor*	1760	x86 AMD	No	6018	2200*	8200*	73%*	x86 AMD, Intel	Yes
HyperDbg*	18967	x86 Intel	No	6018	17800*	23800*	25%*	x86 AMD, Intel	Yes

Figure 6. Porting status of several hypervisor-based open-source research efforts as XMHF hypapps. Note (*) the development of HyperDbg, XTRec and SecVisor is sufficiently advanced to enable estimation of their final sizes via manual inspection of their existing sources and differentiation between the hypervisor core and hypapp-specific logic

(i.e., without any hypapp), and 2) base overhead imposed by XMHF for a given hypapp.

Our platform is an HP Elitebook 8540p with a Quad-Core Intel Core i7 running at 3 GHz, 4 GB RAM, 320GB SATA HDD and an Intel e1000 ethernet controller, using Ubuntu 12.04 LTS as the guest OS 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.

1) Guest Performance: With the rich single-guest execution model (§IV-A) all platform devices are directly accessed and managed by the guest without any intervention (traps) by XMHF. Further, the XMHF runtime gets control only when a configured guest event is explicitly triggered (§IV-B1). Thus, when a well-behaved legacy guest runs, the performance overhead is exclusively the result of the hardware virtualization mechanisms, particularly the Hardware Page Tables (HPT) and the DMA protection.

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), compilebench (project compilation), and unmodified Apache web server performance. For iozone, we perform the disk read and write benchmarks with 4K block size and 2GB file size. We use the compile benchmark from compilebench. We run Apache on top of 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 7. 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. For I/O application benchmarks, read access to files and network access incurs the highest overhead (40% and 25% respectively). The rest of the benchmarks show less than 10% overhead. We attribute the high compute and I/O benchmark latency to benchmark operations that stress the paging logic involving the HPT and I/O DMA logic involving the DMA access control tables. These overheads are comparable to other general-purpose high-performance hypervisors using hardware virtualization including HPT and DMA protections (§VI-C). We expect these overheads to diminish with newer HPT and DMA protection hardware. In general, for both compute and I/O benchmarks, XMHF with 2MB HPT configuration performs better than XMHF with 4KB HPT configuration.

2) Performance of hypapps: A hypapp built on top of XMHF incurs two basic runtime overheads: (a) when the hypapp is invoked via intercepted guest events (including a hypercall), and (b) when the hypapp quiesces cores in a multi-core system in order to perform HPT updates.

When the hypapp is invoked, the CPU switches from guest to host mode, saving the current guest environment state and loading the host environment state. After the hypapp finishes its task, the CPU switches back to guest mode 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 §IV-B2, XMHF employs CPU-quiescing on SMP platforms to ensure intercept serialization. As XMHF uses the NMI for this purpose (§IV-B2), it results in a performance overhead. We measure this overhead by using a simple hypapp that quiesces all other cores, performs a NOP, and then releases them, all in response to a single guest hypercall event. We use a guest application that invokes the hypercall and measure the round-trip time.

The hypapp overheads on XMHF for both 4K and 2MB HPT configurations, for intercepted guest events and quiescing are on average 10 and 13.9 micro-seconds respectively. We note that these hypapp overheads occur every time a guest event is intercepted. Depending on the hypapp functionality this may happen less frequently (a typical and desirable approach today, as evidenced by the hypapps discussed in §VI-A) or more frequently. In either case, the overheads are chiefly due to the hardware (intercept world-switch and NMI signaling). We expect this to diminish as hardware matures. As these overheads reduce, we could conceivably have hypapps interact with the guest in the same spirit as regular applications interact with OS kernels today.

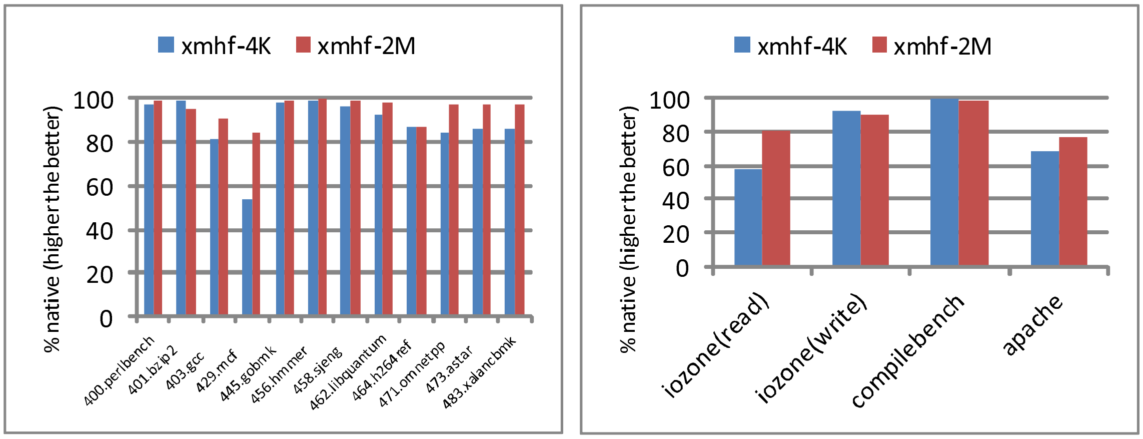

Figure 7. XMHF Application Benchmarks; xmhf-4K = 4K HPT mapping; xmhf-2M = 2MB HPT mapping.

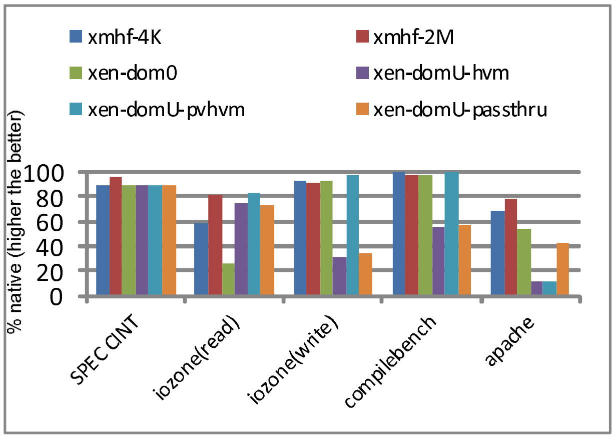

Figure 8. XMHF Performance Comparison with Xen: XMHF and Xen have similar performance for compute-bound and disk I/O-bound applications; XMHF performance is closer to native speed than Xen for network-I/O.

Prog.	OP	SP	VCC	Vars	CLS	T_D	T	M
P	1654	1452	111	437K	1560K	25	76	1.9
P_1^M	1667	1465	116	438K	1561K	27	81	1.9
P_2^M	1668	1466	116	438K	1561K	25	79	1.9
P_3^M	1669	1467	116	438K	1561K	25	79	1.9
P_4^M	1653	1451	117	463K	1668K	25	80	1.9
P_1^L	1679	1477	111	476K	1728K	28	82	1.9
P_2^L	1654	1452	111	437K	1559K	25	79	1.9
P_3^L	1652	1450	111	437K	1559K	25	79	1.9
P_4^L	1634	1441	111	437K	1560K	25	79	1.9
P_5^L	1652	1450	111	437K	1560K	24	78	1.9
P_6^L	1652	1450	111	437K	1560K	25	79	1.9

Figure 9. XMHF verification results with CBMC. OP = number of assignments before slicing; SP = number of assignments after slicing; VCC = number of VCCs after simplification; Vars = number of variables in SAT formula; CLS = number of clauses in SAT formula; T_D = time (sec) taken by SAT solver; T = total time (sec); M = maximum memory (GB)

C. Performance Comparison

We now compare XMHF's performance with the popular Xen (v 4.1.2) hypervisor. 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. We use the compute and I/O-bound application benchmarks as described previously (see §VI-B1). Figure 8 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.

D. Verification Results

We now describe our experiments in verifying DRIVE properties and invariants by model-checking the XMHF implementation. These verification problems are reduced to proving the validity of assertions in a sequential C program P (§V). We discuss our experience in using several publicly available software model checkers to verify P. All experiments were performed on a 2 GHz machine with a time limit of 1800 seconds and a memory limit of 10GB.

1) Experience with CBMC*:* CBMC [22] is a bounded model checker for verifying ANSI C programs. It supports advanced C features like overflow, pointers, and function pointers, and can find bugs such as pointer dereferencing to unallocated memory and array out-of-bounds accesses. It is therefore uniquely suited to verify system software such as XMHF. CBMC is only able to verify programs without unbounded loops; P (XMHF core) complies with this requirement. During verification of P, CBMC automatically sliced away unreachable code and unrolled the remaining (bounded) loops.

The version of CBMC available publicly when we began our experiments was 4.0. This version did not handle two C features that are used in P – function pointers and typecasts from byte arrays to structs. We believe that these features are prevalent in system software in general. We contacted CBMC developers about these issues, and they incorporated fixes in the next public release CBMC 4.1. CBMC 4.1 verifies P successfully.

We also seeded errors in P to create ten additional buggy programs. Four of the buggy programs (P_1^M – P_4^M) contain memory errors that dereference unallocated memory. The remaining six buggy programs (P_1^L – P_6^L) have logical errors that cause assertion violations. In each case, CBMC finds the errors successfully. Table 9 summarizes the overall results for CBMC 4.1. Note that the SAT instances produced are of non-trivial size, but are solved by the back-end SAT solver used by CBMC in about 25 seconds each. Also, about 75% of the overall time is required to produce the SAT instance. This includes parsing, transforming the program to an internal representation (called a GOTO program), slicing, simplification, and generating the SAT formula.

2) Experience with Other Model Checkers: We also tried to verify P and the ten buggy programs with three other publicly available software model checkers that target C code – BLAST [38], SATABS [39], and WOLVERINE [40]. All these model checkers are able to verify programs with loops and use an approach called Counterexample Guided Abstraction Refinement (CEGAR) [41], [42] combined with predicate abstraction [43]. BLAST 2.5 could not parse any of the target programs. In contrast, SATABS 3.1 timed out in all cases after several iterations of the CEGAR loop. On the other hand, WOLVERINE 0.5c ran out of memory in all cases during the first iteration of the CEGAR loop.

VII. Related Work

BitVisor [44] and NoHype [45], [46] are hypervisors that eliminate runtime complexity by running guests with preallocated memory, and direct access to devices. XMHF also advocates the *rich* single-guest execution model (§ IV-A). However, XMHF is designed to provide a common hypervisor core functionality given a particular CPU architecture while at the same time supporting extensions that can provide custom hypervisor-based solutions ("hypapps") for specific functional and security properties. The extensibility of XMHF allows hypapps to be built around it while preserving memory integrity. Xen [47], KVM [29], VMware [48], NOVA [49], Qubes [50] and L4 are general purpose (opensource) hypervisors and micro-kernels which have been used for hypervisor based research [11]–[16], [28]. However, unlike XMHF, they do not present clear extensible interfaces for hypapp developers or preserve memory integrity. Further, complexity arising from device multiplexing and increased TCB make them prone to security vulnerabilities [51]–[55]. OsKit [56] provides a framework for modular OS development. XMHF provides a similar modular and extensible infrastructure for creating hypapps.

A sound architecture [57], [58] is known to be essential for the development of high quality software. Moreover, there has been a body of work in using architectural constraints to not only to drive the analysis of important quality attributes – but also to make such analysis more tractable [59]. Our work reaffirms these ideas, and demonstrates concretely the synergy between – and importance of – architecture and analysis in the context of developing an integrity-protected hypervisor.

The idea of an interface constrained adversary [33], [60] has been used to model and verify security properties of a number of systems. In particular, pinning down the attacker's interface enables systematic and rigorous reasoning about security guarantees. This idea appears in our work as well. Specifically, restricting the attacker's interface to a set of intercept handlers is crucial for the feasibility of DRIVE.

A number of projects have used software model checking and static analysis to find errors in source code, without a specific attacker model. Some of these projects [61]–[63] target a general class of bugs. Others focus on specific types of errors, e.g., Kidd et al. [64] detect atomic set serializability violations, while Emmi et al. [65] verify correctness of reference counting implementation. All these approaches require abstraction, e.g., random isolation [64] or predicate abstraction [65], to handle source code, and therefore, are unsound and/or incomplete. In contrast, we focus on a methodology to develop a hypervisor that achieves a specific security property against a well-defined attacker.

Finally, there has been several projects on verifying security of operating system and hypervisor implementations. Neumann et al. [66], Rushby [67], and Shapiro and Weber [68] propose verifying the design of secure systems by manually proving properties using a logic and without an explicit adversary model. A number of groups [18]–[20] have employed theorem proving to verify security properties of OS implementations. Barthe et al. [69] formalized an idealized model of a hypervisor in the Coq proof assistant and Alkassar et al. [70], [71] and Baumann et al. [72] annotated the C code of a hypervisor and utilized the VCC [21] verifier to prove correctness properties. Approaches based on theorem proving are applicable to a more general class of properties, but also require considerable manual effort. For example, the seL4 verification [20] shows a form of functional correctness, essentially relating the C implementation with a high level specification and required several man years effort. Since the high-level specification does not include an explicit adversary that is trying to break memory integrity, the verification does not imply the security property of interest to us. In contrast, our approach is more automated but we focus on a specific security property (memory integrity) with an explicit adversary model. However, since we do not verify full functional correctness, we cannot, for example, claim that our system will not crash.

VIII. Conclusion and Future Work

We propose an eXtensible and Modular Hypervisor Framework (XMHF) which strives to be a comprehensible and flexible platform for building hypervisor applications ("hypapps"). XMHF is based on a design methodology that enables automated verification of hypervisor memory integrity. In particular, the automated verification was performed on the actual source code of XMHF – consisting of 5208 lines of C code – using the CBMC model checker We believe that XMHF provides a good starting point for research and development on hypervisors with rigorous and "designed-in" security guarantees. Given XMHF's features and performance characteristics, we believe that it can significantly enhance (security-oriented) hypervisor research and development.

One direction for future work is modular verification of the XMHF core composed with hypapps. Another direction is to extend DRIVE to other security properties such as secrecy. A challenge here is that for such a property, the attacker's interface is much less well-defined compared to memory integrity, due to the possibility of covert channels etc. Yet another direction is to include support for concurrent execution within XMHF and the hypapps. The question here is whether, or how, we can still guarantee memory integrity.

Acknowledgements. We thank our shepherd, William Enck, for his help with the final version of this paper, as well as the anonymous reviewers for their detailed comments. We also want to thank Adrian Perrig, Virgil Gligor and Zongwei Zhou for stimulating conversations on XMHF. This work was partially supported by NSF grants CNS-1018061, CCF-0424422, CNS-0831440, and an AFOSR MURI on Science of Cybersecurity. Copyright 2012 Carnegie Mellon University and IEEE[3].

References

[1] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar, "Towards verifiable resource accounting for outsourced computation," in *Proc. of ACM VEE*, 2013.

[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 *Proc. of TRUST*, Jun. 2012.

[3] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with hyperlock," in *Proc. of EuroSys 2012*.

[4] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Proc. of IEEE S&P*, May 2012.

[5] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. of SOSP*, 2011.

[6] A. Vasudevan, N. Qu, and A. Perrig, "Xtrec: Secure real-time execution trace recording on commodity platforms," in *Proc. of IEEE HICSS*, Jan. 2011.

[7] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. of IEEE S&P*, May 2010.

[8] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and transparent analysis of commodity production systems," in *Proc. of IEEE/ACM ASE 2010*.

[9] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. of USENIX Security Symposium*, 2008.

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

[11] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity os from untrusted extensions," in *Proc. of of NDSS 2011*.

[12] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth, "Reducing TCB complexity for security-sensitive applications: Three case studies," in *Proc. of EuroSys*, 2006.

[13] R. Ta-Min, L. Litty, and D. Lie, "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable," in *Proc. of SOSP*, 2006.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. of of ACM CCS 2008*.

[15] D. Quist, L. Liebrock, and J. Neil, "Improving antivirus accuracy with hypervisor assisted analysis," *J. Comput. Virol.*, vol. 7, no. 2, May 2011.

[16] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proc. of ACM CCS*, 2009.

[17] 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 *Proc. of ASPLOS*, Mar. 2008.

[18] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Communications of the ACM (CACM)*, vol. 23, no. 2, 1980.

[19] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. D. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *Proc. of ACM CCS*, 2006.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proc. of SOSP*, 2009.

[21] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A Practical System for Verifying Concurrent C," in *Proc. of TPHOLs*, 2009.

[22] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Proc. of TACAS*, 2004.

[23] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C," 2011.

[24] Advanced Micro Devices, "AMD64 architecture programmer's manual: Volume 2: System programming," AMD Pub-

[3] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0000090

lication no. 24594 rev. 3.11, Dec. 2005.

[25] ARM Limited, "Virtualization extensions architecture specification," http://infocenter.arm.com, 2010.

[26] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity principles, implementations and applications," *TISSEC*, vol. 13, no. 1, 2009.

[27] L. Jia, D. Garg, and A. Datta, "Compositional security for higher-order programs," Carnegie Mellon University, Tech. Rep. CMU-CyLab-13-001, 2013, online at http://www.andrew.cmu.edu/user/liminjia/compositional.

[28] 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 *Proc. of OSDI 2010*.

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

[30] P. Karger and D. Safford, "I/O for virtual machine monitors: Security and performance issues," *IEEE Security and Privacy*, vol. 6, no. 5, 2008.

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

[32] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. of IEEE S&P*, 2010.

[33] A. Datta, J. Franklin, D. Garg, and D. Kaynar, "A logic of secure systems and its application to trusted computing," in *Proc. of IEEE S&P*, 2009.

[34] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The static driver verifier research platform," in *Proc. of CAV*, 2010.

[35] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bmc of concurrent software," *CoRR*, vol. abs/1301.1629, 2013.

[36] J. Franklin, S. Chaki, A. Datta, and A. Seshadri, "Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size," in *Proc. of IEEE S&P*, 2010.

[37] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan, "Parametric Verification of Address Space Separation," in *Proc. of POST*, 2012.

[38] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *Proc. of POPL*, 2002.

[39] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *Proc. of TACAS*, 2005.

[40] D. Kroening and G. Weissenbacher, "Interpolation-Based Software Verification with Wolverine," in *Proc. of CAV*, 2011.

[41] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, 2003.

[42] T. Ball and S. K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," in *Proc. of SPIN*, 2001.

[43] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in *Proc. of CAV*, 1997.

[44] 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 *Proc. of ACM SIGPLAN/SIGOPS VEE 2009*.

[45] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proc. of ACM CCS*, 2011.

[46] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *Proc. of ISCA*, 2010.

[47] 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 *Proc. of SOSP*, 2003.

[48] VMware Corporation, "VMware ESX, bare-metal hypervisor for virtual machines," http://www.vmware.com, Nov. 2008.

[49] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proc. of the Eurosys*.

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

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

[52] "Elevated privileges," CVE-2007-4993, 2007.

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

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

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

[56] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The flux oskit: A substrate for os and language research," in *Proc. of ACM SOSP 1997*.

[57] M. Shaw and D. Garlan, *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[58] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison Wesley, 2003.

[59] K. Wallnau, "Volume III: A Technology for Predictable Assembly from Certifiable Components," Software Engineering Institute, Carnegie Mellon University, Technical report CMU/SEI-2003-TR-009, 2003.

[60] D. Garg, J. Franklin, D. K. Kaynar, and A. Datta, "Compositional System Security with Interface-Confined Adversaries," *ENTCS*, vol. 265, 2010.

[61] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses."

[62] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in *Proc. of CCS*, 2002.

[63] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *Proc. of OSDI*, 2004.

[64] N. Kidd, T. Reps, J. Dolby, and M. Vaziri, "Finding Concurrency-Related Bugs Using Random Isolation," in *Proc. of VMCAI*, 2009.

[65] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar, "Verifying Reference Counting Implementations," in *Proc. of TACAS*, 2009.

[66] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson, "A provably secure operating system: The system, its applications, and proofs." SRI International, Tech. Rep., 1980.

[67] J. M. Rushby, "Design and Verification of Secure Systems," in *Proc. of SOSP*, 1981.

[68] J. S. Shapiro and S. Weber, "Verifying the EROS Confinement Mechanism," in *Proc. of IEEE S&P*, 2000.

[69] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "Formally Verifying Isolation and Availability in an Idealized Model of Virtualization," in *Proc. of FM*, 2011.

[70] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova, "Automated Verification of a Small Hypervisor," in *Proc. of VSTTE*, vol. 6217, 2010.

[71] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. J. Paul, "Verifying shadow page table algorithms," in *Proc. of FMCAD*, 2010.

[72] C. Baumann, H. Blasum, T. Bormer, and S. Tverdyshev, "Proving memory separation in a microkernel by code level verification," in *Proc. of AMICS*, 2011.