Design, Development and Automated Verification of an Integrity-Protected Hypervisor

Sagar Chaki, Amit Vasudevan, Limin Jia, Jonathan McCune, and Anupam Datta

July 16, 2012

CMU-CyLab-12-017

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Design, Development, and Automated Verification of an Integrity-Protected Hypervisor*

Sagar Chaki Amit Vasudevan Limin Jia Jonathan McCune Anupam Datta

Carnegie Mellon University, Pittsburgh, PA, USA

chaki@sei.cmu.edu amitvasudevan@acm.org liminjia@cmu.edu
jonmccune@cmu.edu danupam@cmu.edu

ABSTRACT

Hypervisors are a popular mechanism for implementing software virtualization. Since hypervisors execute at a very high privilege level, they must be secure. A fundamental security property of a hypervisor is *memory integrity* – the hypervisor's memory must not be modified by software running at a lower privilege level. In this paper, we present a methodology – called DRIVE – for designing, developing, and verifying hypervisors to ensure memory integrity. DRIVE combines the power of architectural constraints (captured by a set of system properties and verification conditions) with that of formal analysis (used to discharge the verification conditions). We prove that any hypervisor satisfying the DRIVE properties and verification conditions has memory integrity. We validate DRIVE by using it to develop a hypervisor called XMHF for multi-core systems. In particular, we show how to ensure the DRIVE properties in XMHF by combining hardware virtualization support with design and development decisions. We also show how to discharge the DRIVE verification conditions on XMHF using the CBMC model checker. CBMC verified XMHF's implementation – about 4700 lines of C code – in about 80 seconds using less than 2GB of RAM.

General Terms

Security; Verification; Hypervisor; Integrity; Model Checking; Design

1. INTRODUCTION

Hypervisors [54] are increasingly used on modern computing platforms, including desktops, servers, mobile and cloud computing [25, 55]. A hypervisor is a mechanism to help implement a virtual computing platform. It sits between the hardware and one or more "guest" operating systems, presenting to the guests a virtual operating platform and managing their execution. Hypervisors have proved to be an effective means to improve hardware utilization, reduce power and cooling costs, and streamline backup, recovery, and data center management, thus leading to their widespread adoption in practice.

However, from a system security perspective, hypervisors are yet another maximally privileged software component in the trusted computing base. While they are always designed and implemented to meet stringent performance and feature requirements, formal assurances about their security and functional correctness properties are often ignored. In several instances, vulnerabilities have been reported in deployed hypervisors (e.g., [1–7, 46, 70]), which have subsequently been patched. Recognizing that iterative patching of discovered vulnerabilities is both expensive and dangerous as hypervisors and related systems software become more complex and tightly interwoven into critical software systems, there have been several efforts at verifying security-relevant properties of the source code of these systems [22, 40, 45, 68]. However, most of these efforts have required significant *manual* effort.

Against this background, we make two contributions. Our first contribution is a methodology for designing, developing, and *automatically* verifying hypervisors to ensure *memory integrity*. Roughly, memory integrity means that the hypervisor memory is only modified by software running in privileged mode. In particular, this implies that guests are unable to modify hypervisor memory directly. We focus on memory integrity because it is a key security property for virtualized systems. Without memory integrity, portions of the hypervisor that manage the isolation of memory pages between guests are open to malicious modifications, thereby allowing one guest to read and modify the code or data of another guest or the hypervisor itself. Memory integrity is therefore necessary for other important security goals, such as data secrecy and availability of the hypervisor as well as guests.

We call our methodology DRIVE – "Designing hypervisors for Rigorous Integrity VErification". Intuitively, DRIVE is composed of a set of hypervisor properties, a set of verification conditions required to be true of the hypervisor, and a proven claim that the properties and verification conditions entail the hypervisor's memory integrity. Thus, DRIVE combines architectural constraints (expressed by the system properties and verification conditions) with that of automated formal analysis (used to discharge the verification conditions). Specifically, DRIVE stipulates the following properties:

1. The hypervisor architecture is modular (property **MOD**)—it includes an initialization function that runs when the system starts executing and a set of *intercept handlers* that are invoked when certain events are caused by guests (e.g., I/O operations, interrupts) and devices (e.g., direct memory access). Following prior work in the security literature [32,49], we define the capabilities of the adversary against the hypervisor in terms of the hypervisor interfaces (i.e., the intercept handlers) that it can invoke with arbitrary inputs.
2. The hypervisor includes a mandatory access control mecha-

*This work was partially supported by NSF grants CCF-0424422, CNS-1018061 and CNS-0831440, by ARO contracts W911NF0910273 and DAAD-190210389 to Carnegie Mellon CyLab, and 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.

nism to mediate access to hypervisor memory in unprivileged mode (property **MAC**).

3. The hypervisor has control flow integrity [8], i.e., the control flow during the execution of H always respects its source code (**CFI**).
4. The initialization function executes atomically in a single-threaded environment and each intercept handler also executes atomically in a single-threaded environment (**ATOM**).

In addition, DRIVE includes two verification conditions that state that memory integrity holds after the initialization function executes and is preserved by every intercept handler. Our main result (Theorem 1) is that any hypervisor satisfying the DRIVE properties and verification conditions has memory integrity.

Even though a virtualized system is inherently concurrent – e.g., guests are multi-threaded and execute on multiple cores – an important feature of DRIVE is that discharging its verification conditions requires us to verify only sequential programs. The main reason behind this is that the system properties mandated by DRIVE enable us to "sequentialize" (see Definition 1) the semantics of the target virtualized system. Verification of sequential programs is recognized to be more tractable than that of concurrent programs. Thus, DRIVE concretizes the idea [15,59,69] that architectural constraints enable more effective analysis for ensuring quality attributes. Further details are presented in Section 3.

Our second contribution is the development of a hypervisor called XMHF [65] using DRIVE. In particular, we show how the DRIVE system properties are ensured in XMHF by a combination of hardware virtualization primitives, and design and development decisions. In addition, we show how the DRIVE verification conditions are discharged on XMHF using the software model checker CBMC [19]. CBMC discharged the verification conditions on the entire XMHF runtime – about 4700 lines of C code – in about 80 seconds using less than 2GB of RAM (see Section 4.6 for more details).

The verification of XMHF was not only automated, but also *development compatible*. More specifically, it was engineered to be repeatable with minimal effort as the implementation of XMHF was modified. In particular, this means that the verification required no manually supplied invariants or annotations. Development compatibility was extremely useful as XMHF was verified repeatedly and routinely (in fact, as part of its build process) during development. Further details about the engineering of XMHF verification and its development compatibility are presented in Sections 4.3–4.5.

Indeed, our experience with XMHF has led to the belief that DRIVE is compatible with – and would aid – cost-effective software maintenance. Maintainability is a desirable quality attribute of system software, such as hypervisors, which typically have long life spans. History suggests that disentangling development and evaluation for desired security properties of software [17, 23, 24, 28, 33, 34, 60, 62, 63] is extremely expensive. In the case of general software assurance, maintainability is often aided by a set of standards [64] for evaluating whether a patch or other change warrants full re-evaluation. We expect that DRIVE would play a similar role in the context of hypervisor integrity.

From here on, unless otherwise mentioned, we use the term integrity to refer to memory integrity. The rest of this paper is organized as follows. Section 2 provides background on hypervisors, hardware virtualization primitives, and memory integrity. Section 3 presents the DRIVE methodology. In Section 4 we describe our experience in applying DRIVE to develop and verify XMHF in a development compatible manner. In Section 5, we survey related work. Finally, we discuss important issues raised by this research and conclude in Section 6.

2. BACKGROUND

A hypervisor is a popular hardware virtualization technique that allows multiple operating systems, termed *guests*, and virtual devices to run concurrently on a host computer. It is so named because it is conceptually one level higher than a supervisory (OS kernel) program. The hypervisor presents to the guests and devices a virtual operating platform and manages their execution. In general, the guests consist of multiple instances of a variety of operating systems sharing the virtualized hardware resources. The guests and devices are untrusted and constitute the attacker.

2.1 Virtualization Primitives

We focus on hypervisors that rely on certain hardware virtualization primitives. These primitives are supported by current x86 computing platforms [11, 43], and are also making their way on embedded ARM architectures [12]. In particular, we are interested in hardware virtualization primitives that enable the following features:

- The CPU executes in two overarching modes: (a) *host-mode* (or privileged mode) – where the hypervisor executes, and (b) *guest-mode* (or unprivileged mode) – where the guests execute. The privileged and unprivileged modes have separate address spaces and CPU registers.
- At system boot time, the hypervisor is able to execute a designated piece of code before the attacker has access to system memory. This is used by the hypervisor to correctly initialize memory protection mechanisms.
- The execution state of each guest is maintained in a separate data structure. This is important for guest event handling, as described next.
- The hypervisor is able to associate *intercept handlers* with certain events caused by the attacker. Specifically, these events are caused by guests (e.g., instructions, I/O operations, exceptions and interrupts) and devices (e.g., direct memory access). The hardware ensures that whenever such an event e occurs, the following sequence of actions occur:
  1. The execution state of each guest is saved in its own data structure.
  2. CPU execution is switched to privileged mode.
  3. The intercept handler for e is executed.
  4. CPU execution is switched back to unprivileged mode.
  5. Execution state of each guest is restored and guest execution is resumed.

Figure 1 shows a high level architectural view of a virtualized system that relies on the virtualization features presented above. Popular commercial and open-source hypervisors – e.g., VMware ESX/ESXi, Hyper-V, KVM, and VirtualBox – adhere to this architectural view. Note that memory access occurs in three ways: (i) during hypervisor initialization; (ii) by guests and devices; and (iii) by intercept handlers triggered by the guests and devices.

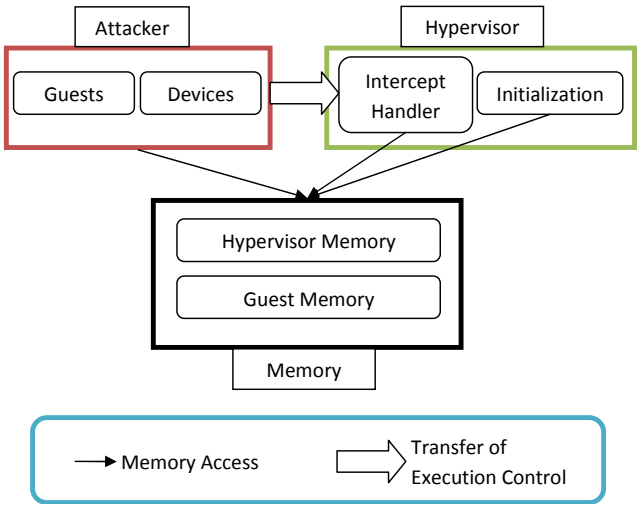

Figure 1: High level view of a hypervisor.

2.2 Integrity Protection

Recall that we focus on integrity as our security objective. Note that in Figure 1, we differentiate between two types of memory – hypervisor memory denoted by a set of addresses $\mathcal{M}$, and guest memory. Note that $\mathcal{M}$ refers to both hypervisor code and data. Integrity means that hypervisor memory is never modified by attacker code. Since attacker code always runs in unprivileged mode, integrity is ensured if hypervisor memory is only modified by code executing in privileged mode.

More specifically, 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 intercept handlers). Further, integrity requires that neither hypervisor code nor data can be directly accessed via Direct Memory Accesses (DMA) by devices. Note that shared memory pages between the hypervisor and guests – which are writable by guests – are considered to be part of guest memory. Modification of these pages by the attacker, therefore, does not violate integrity. Finally, we assume that the hardware behaves in accordance with its specification.

3. THE DRIVE METHODOLOGY

In this section, we present the DRIVE methodology for hypervisor design and development. We consider a virtualized system $V = (H, A, \mathcal{M})$, where H is the hypervisor, A is the attacker representing malicious guests and devices, and $\mathcal{M}$ is the hypervisor memory. As mentioned before, DRIVE consists of a set of system properties and verification conditions, together with an argument the these properties and verification conditions imply hypervisor integrity. We first present the DRIVE properties and verification conditions in detail. We end this section with the argument – see Theorem 1 – that any hypervisor satisfying the DRIVE properties and verification conditions has memory integrity.

3.1 DRIVE Properties

Recall that V executes in two modes – privileged and unprivileged. The DRIVE methodology mandates four properties on V. The first property ensures mandatory control of accesses to hypervisor memory. It is expressed as two sub-properties:

- (**MAC**)(a) H uses an access control mechanism to control access to memory in unprivileged mode, and stores all state related to the access control mechanism in $\mathcal{M}$; (**MAC**)(b) in unprivileged mode, the hardware ensures that all access to memory is subjected to the access control mechanism.

The remaining three properties impose restrictions on the hypervisor's implementation and its response to the attacker's actions.

- (**CFI**) The hypervisor has control flow integrity [8], i.e., the control flow during the execution of H always respects its source code.

- (**MOD**) Initialization is implemented by a function $init()$ and the intercept handlers are implemented by functions $ih_1(), \ldots, ih_k()$.

- This property ensures the atomicity of initialization and intercept handling. It is expressed by two sub-properties: (**ATOM**)(a) at the start of V's execution, $init()$ runs completely in a single-threaded environment before any other code executes; (**ATOM**)(b) the intercept handlers $ih_1(), \ldots, ih_k()$ always execute in a single-threaded environment.

Note that **ATOM** requires only privileged (i.e., hypervisor) code to be serialized. Unprivileged code is able to exercise all available cores and spawn as many threads as necessary. As recent CPUs supporting hardware virtualization are designed to minimize traps to the hypervisor when executing in unprivileged mode, **ATOM** does not unacceptably degrade the hypervisor's performance.

Figure 2 gives an architectural view of a virtualized system developed via DRIVE. Note that it is a refined version of Figure 1. In particular, it shows that all memory accesses by the attacker are mediated by the access control mechanism implemented in hardware. The mechanism checks whether the memory access is allowed by the access control mechanism state, and either lets it proceed, or triggers the appropriate intercept handler in H.

3.2 Sequentialization

Properties **CFI**, **MOD** and **ATOM** lead to a natural sequentialization of V's execution, denoted by $Seq(V)$, and shown in Figure 3. Intuitively, the first step after "power on" is initialization, during which the memory access table is set up. Subsequently, code executes either:

- in unprivileged mode by A, specifically, by guest code and direct memory access (DMA) by devices; or

- in privileged mode by H, specifically, by intercept handlers triggered by A.

Given two sequential programs f and g, let $f + g$ denote the sequential program that non-deterministically executes either f or g. Clearly $+$ is commutative and associative. Also, for any function f, let $f(*)$ denote the execution of f under an arbitrary calling context. Then $Seq(V)$ is defined formally as follows:

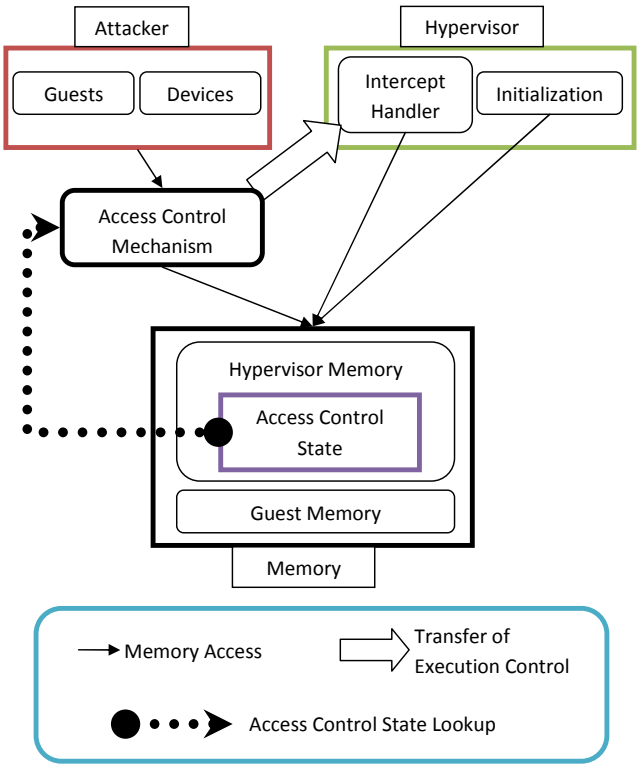

Figure 2: High level view of a virtualized system developed via DRIVE.

DEFINITION 1 (SEQUENTIALIZATION). *$Seq(V)$ is given by the following sequential program:*

$$init(*);\ while(true)\ \{\ A() + ih_1(*) + \cdots + ih_k(*)\ \}$$

where $A()$ corresponds to the execution of arbitrary attacker code.

3.3 Verifying Integrity

We now use $Seq(V)$ to present our algorithm for verifying the integrity of V. Our key result, expressed in Theorem 1, is that verifying the integrity of V reduces to discharging a set of verification conditions over $init(*)$ and $ih_1(*), \ldots, ih_k(*)$. We begin with the definition of integrity protected memory.

DEFINITION 2 (INTEGRITY PROTECTED MEMORY). *The hypervisor memory is said to be integrity protected iff the following condition – denoted by $\varphi(\mathcal{M})$ – holds:*

$$\varphi(\mathcal{M}) \quad \equiv \quad \textit{all memory addresses in } \mathcal{M} \textit{ are designated read-only in unprivileged mode}$$

THEOREM 1. *V is integrity protected iff the following two verification conditions hold:*

1. **(VC1)** *$init(*)$ ensures $\varphi(\mathcal{M})$.*
2. **(VC2)** *for $i \in [1, k]$, $ih_i(*)$ preserves $\varphi(\mathcal{M})$.*

Proof Sketch. Recall that integrity means that the contents of hypervisor memory $\mathcal{M}$ are only modified in privileged mode. The forward implication is trivial since, if either **VC1** or **VC2** does not hold, then there is an input x such that $init(x)$ or $ih_i(x)$ leads to a violation of $\varphi(\mathcal{M})$. Subsequently, unprivileged code is free to modify $\mathcal{M}$.

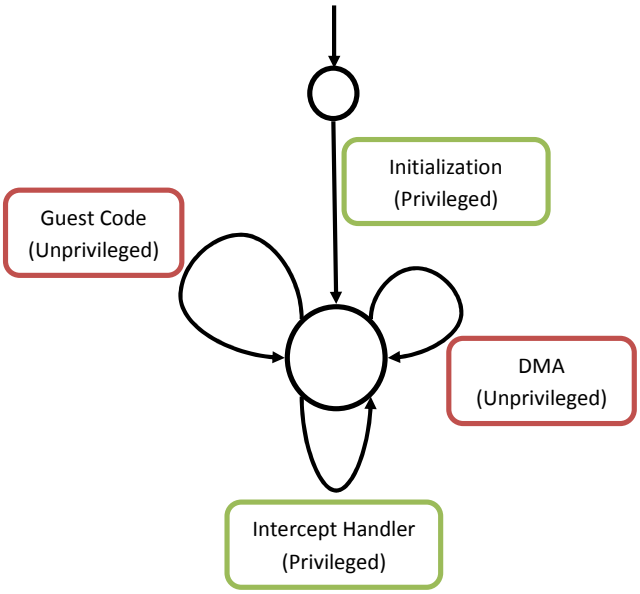

Figure 3: Life cycle of a DRIVE virtualized system from the point of view of memory accesses.

For the reverse implication, recall the definition of $Seq(V)$ from Definition 1. Note that **MAC** implies that the execution of the attacker code $A()$ preserves $\varphi(\mathcal{M})$. Indeed, since the access control mechanism state is itself part of $\mathcal{M}$, any attempt by $A()$ to violate $\varphi(\mathcal{M})$ leads to the triggering of an intercept handler.

Therefore, if **VC1** and **VC2** holds, then from Definition 1, we know that $\varphi(\mathcal{M})$ is an inductive invariant of $Seq(V)$. This, together with **MAC**, implies that V is integrity protected. □

Note that **MAC** is used directly in the proof of Theorem 1. The remaining three properties (**CFI**, **MOD**, and **ATOM**) are required to define $Seq(V)$ (Definition 1), on which the proof of Theorem 1 relies as well. Thus, all DRIVE properties are necessary for the soundness of our approach. Note that **VC1** and **VC2** can be discharged by verifying a sequential program, even though the virtualized system itself is concurrent. The key here is of course properties **CFI**, **MOD**, and **ATOM**, which enable us to define the semantics of V as a sequential program and subsequently prove Theorem 1.

4. USING DRIVE TO DEVELOP XMHF

In this section, we report on our experience in using DRIVE to develop XMHF [65]. In particular, we show how the DRIVE properties guide the design and development of XMHF, and how the DRIVE verification conditions are reduced to assertions that are then automatically model checked on XMHF's source code.

4.1 XMHF Background

Architecturally, XMHF consists of an XMHFCORE and an HYPAPP [65]. The XMHFCORE contains the core hypervisor functionality (e.g., platform initialization, multi-core support, memory and DMA protections) while the HYPAPP *extends* this core functionality

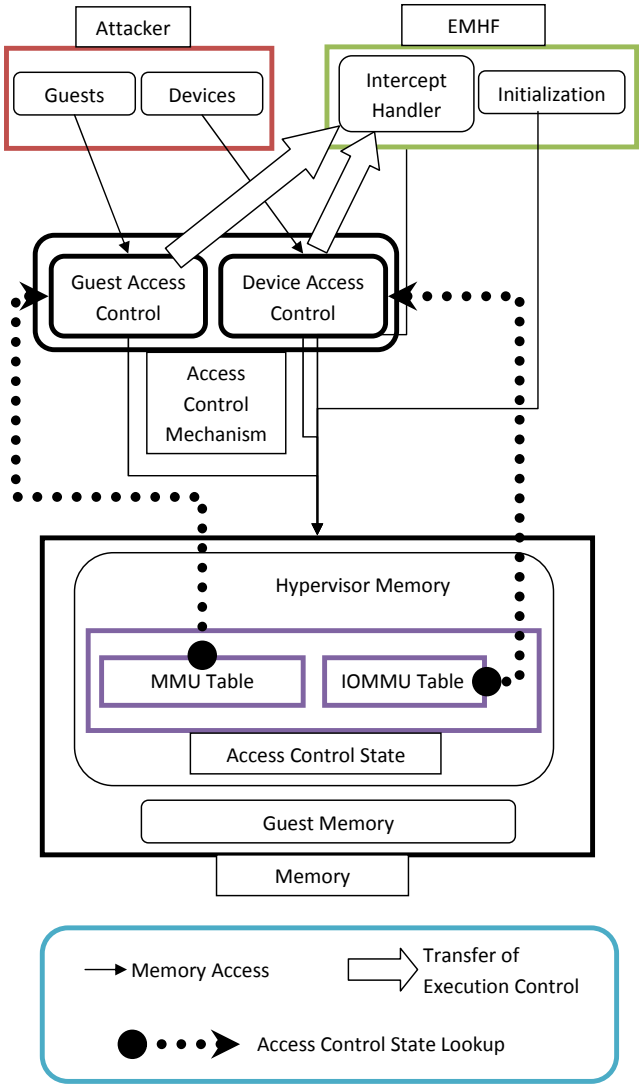

Figure 4: High level view of a virtualized system based on XMHF.

to implement a customized hypervisor. XMHF runs on commodity hardware-virtualized x86 platforms (Intel and AMD).

XMHF supports a single guest. If the underlying hardware is multi-core, the guest is able to utilize as many cores as it needs. All interrupts are *passthru* to the guest. This means that the guest handles all these interrupts directly without intervention from XMHF, resulting in optimal system performance. Note that this hypervisor execution model of a single-guest with *passthru* interrupts resonates with mechanisms employed by several recent research efforts in the ad-hoc hypervisor space such as CloudVisor [72], TrustVisor [50], Lockdown [66], XTRec [67], SecVisor [57], Proxos [61] and Nizza [39]. XMHF therefore provides a common base which helps in realizing these existing hypervisor solutions as HYPAPP instances, while also providing memory integrity.

Figure 4 shows a high level view of a virtualized system based on XMHF. Note that Figure 4 further refines Figure 2 by incorporating details specific to XMHF. In particular, the access control mechanism state consists of two tables – the Memory Management Unit (MMU) table [1], and the Input Output Memory Management Unit (IOMMU) table.

Each core on a hardware-virtualized x86 CPU executes in either *host-mode* (where the hypervisor executes) or *guest-mode* (where the guest executes). In either mode, the hardware uses the IOMMU table to determine if a DMA transfer by a particular device is allowed. If a violation of the IOMMU permissions is observed, the hardware disallows the requested DMA. In contrast, the hardware enforces MMU table access control *only in guest mode*. In particular, the hardware ensures that all memory accesses by guest instructions go via a two-level translation in the presence of the MMU table. 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 MMU table. If the access requested by the guest violates the permissions stored in the MMU table, the hardware triggers an `npf` exception.

4.2 Mechanisms to Ensure DRIVE Properties

At system startup, XMHF is loaded via a *dynamic-launch* operation [42] – a feature available on commodity x86 CPUs. Using dynamic-launch ensures that the XMHF loader code executes in a hardware-protected environment, which in turn transfers control to the XMHFCORE runtime. The runtime initializes the hypervisor memory such that $\varphi(\mathcal{M})$ (see Definition 2) holds, switches the CPU execution to *guest-mode*, and starts executing the guest. We now describe how XMHF ensures **MAC**, **CFI**, **MOD**, and **ATOM**.

4.2.1 Ensuring MAC

For **MAC** (a), XMHF uses the MMU table and the IOMMU table to store memory access permissions for guests and devices, respectively. In addition, XMHF ensures that both MMU and IOMMU tables reside in the hypervisor memory $\mathcal{M}$. Thus, as shown in Figure 4, the access control mechanism is logically partitioned into two sub-mechanisms, one for guests (that uses the MMU table) and the other for devices (that relies on the IOMMU table).

XMHF ensures **MAC** (b) by a combination of system initialization, runtime intercept handling, and hardware semantics. Recall that IOMMU and MMU tables can only be initialized and activated by software running on the CPU in host-mode. However, the IOMMU table access control protections are enforced by the hardware for all devices in the system in both guest and host modes. In contrast, MMU access control protections are enforced by a CPU core only when the core is operating in guest-mode. On x86 (and ARM) platforms, only one core – called the boot-strap processor (BSP) – is enabled when the system starts. The other cores are placed in a halted state until activated by software running on the BSP. The BSP starts up devoid of any memory protections. During its initialization, XMHF switches the BSP to host-mode and sets up the IOMMU table. This ensures that IOMMU access control remains enabled in all future system states. XMHF then activates the remaining cores in the system and switches them to host-mode as well. Next, XMHF sets up the MMU table on all the cores and switches the BSP to guest-mode to boot the guest operating system; the remaining cores idle in host-mode. Finally, XMHF uses intercept handling to ensure that the remaining cores are switched to guest-mode before they execute guest code. This scheme ensures that MMU access control is always enabled for all CPU cores in the

[1]The MMU table is called the "Nested Page Table" and the "Extended Page Table" on AMD and Intel platforms, respectively.

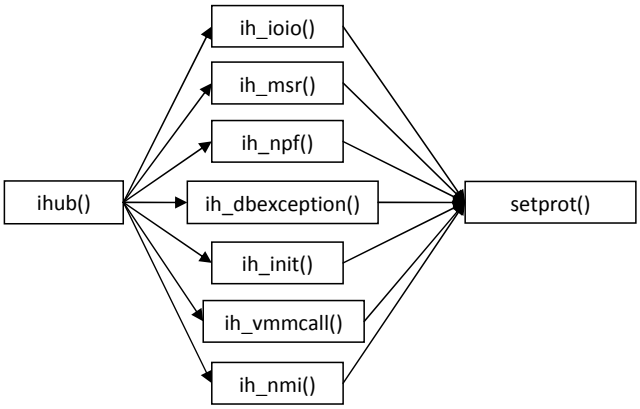

Figure 5: Partial call graph of the top-level intercept handler $ihub()$ in XMHF. The function `setprot()` is used to make all changes to the MMU and IOMMU tables.

system. The exact details behind this process are presented later in Section 4.3.

4.2.2 *Ensuring* CFI

We assume that XMHF satisfies **CFI**, i.e., it has control flow integrity. Since XMHF is implemented in C, we believe that existing techniques [8] are capable of discharging this assumption.

4.2.3 *Ensuring* MOD

MOD is ensured by organizing the source code of XMHF to make the implementations of $init()$ and $ih_1(), \ldots, ih_k()$ modular. We now present this step in more detail.

The name of the $init()$ function in XMHF is `emhf_runtime_main()`. This function first performs required platform initialization, initializes memory such that $\varphi(\mathcal{M})$ holds, then switches the *boot-strap processor* (BSP) to guest-mode before starting the guest.

XMHF has a single top-level intercept handler function called `emhf_parteventhub_intercept_handler`. For brevity, we refer to this function as $ihub()$. This function is called whenever one of the following seven intercepts[2] is triggered:

- `ioio`- for I/O port interception;
- `msr` - for trapping accesses to critical CPU model specific registers;
- `npf`- for handling nested page faults;
- `dbexception`- for ensuring guest-mode execution (see Section 4.3);
- `init` - for handling guest shutdown and restarts;
- `vmmcall` - for handling guest hypercalls; and
- `nmi`- for ensuring **ATOM** (see Section 4.2.4).

[2]Current x86 hardware allow intercepting additional intercepts (e.g., intercept on task-switch, execution of certain instructions etc.). However, these seven intercepts suffice to ensure properties **MAC**, **CFI**, **MOD**, and **ATOM**, and to discharge **VC1** and **VC2**.

The arguments of $ihub()$ indicate the actual interrupt that was triggered. Based on the value of these arguments, $ihub()$ executes an appropriate sub-handler. For simplicity, we refer to the sub-handler that handles intercept e as $ih\_e()$. Figure 5 shows a partial call-graph for $ihub()$ highlighting the sub-handlers, and a special function `setprot()` used to discharge **VC1** on XMHF (see Section 4.4 for more details).

As mentioned before, the `npf` intercept indicates a violation of MMU table permissions by a guest memory access. The `nmi` intercept is used to ensure **ATOM** (b), and the `dbexception` intercept is used to ensure **MAC** (b) on multi-core hardware. These are discussed in more detail later.

4.2.4 *Ensuring* ATOM

ATOM (a) is ensured by initially boot-strapping XMHF using a *dynamic-launch* operation. Dynamic-launch is a capability on current x86 (AMD and Intel) platforms that allows an arbitrary piece of code to execute in isolation from everything else on the system except for the CPU, memory, and chipset. The use of dynamic-launch and the launched code is securely recorded in the Trusted Platform Module (TPM) [36]. Further, the environment after dynamic-launch does not allow any interrupts or asynchronous executions.

ATOM (b) is ensured by XMHF using a mechanism called *core quiescing*, which is implemented as follows. Suppose an intercept e is triggered on a specific core C. If e is `nmi`, then $ih\_e()$ is implemented to be an idle loop. Otherwise, the first thing done by $ih\_e()$ is to send a Non-Maskable Interrupt (NMI) to all cores other than C. Since the NMI cannot be masked, this causes all these other cores to execute $ih\_\texttt{nmi}()$. Since $ih\_\texttt{nmi}()$ is an idle loop, in effect, all these other cores stall. $ih\_e()$ then handles e properly, and finally reactivates the other cores before resuming the guest.

4.3 Verifying Guest-Mode Execution

Recall that to ensure **MAC** (b), we must ensure that each core is set to execute in guest-mode before it executes any attacker code (Section 4.2.1). As mentioned before, on a single core CPU, XMHF's $init()$ code switches the BSP core to guest-mode before booting the OS. For a multi core CPU XMHF then activates the remaining cores in the system and switches them to host-mode which then idle within XMHF. Finally, XMHF uses the hardware's multi-core bring-up logic and intercept handling to ensure that the remaining cores are also switched to guest-mode before they execute guest code. We now describe this process in more detail in the context of commodity x86 platforms.

To bring a new core C online, the guest sends a startup inter-processor interrupt (SIPI) to C. However, on current x86 platforms, the default operating mode of C in response to a SIPI does not include any memory protections. Therefore, XMHF intercepts the SIPI and switches C to guest-mode before handing back execution to the guest. For x86 platforms, a SIPI interrupt is delivered to C via the CPU Local Advanced Programmable Interrupt Controller (LAPIC). Specifically, the LAPIC has an Interrupt Control Register (ICR) that is used to deliver the SIPI to C.

To support both Intel and AMD x86 platforms, XMHF uses a unified scheme to intercept guest multi-core activation. On both Intel and AMD hardware-virtualized platforms, the LAPIC registers are accessed via memory-mapped I/O. The memory-mapped I/O region typically encompasses a single physical memory page. XMHF

```
//top-level intercept handler
//x specifies triggered intercept
void ihub(int x)
{
#ifdef VERIFY
  int pre_npfe = NPFELAPIC_TRIGGERED(x);
  int pre_dbe = DBE_TRIGGERED(x);
#endif
  ...
  //main body of ihub
  ...
#ifdef VERIFY
  assert (!pre_npfe || GUEST_TRAPPING(x));
  assert (!pre_dbe || CORE_PROTECTED(x));
#endif
}
```

Figure 6: Outline of $ihub()$, which is the top-level intercept handler function. Macro `DBE_TRIGGERED(x)` evaluates to true iff `x` indicates that the `dbexception` intercept was triggered. Macro `NPFELAPIC_TRIGGERED(x)` evaluates to true iff `x` indicates that the `npf` intercept was triggered in response to the guest accessing the LAPIC memory-mapped I/O page. Macro `GUEST_TRAPPING(x)` evaluates to true iff the core identified by `x` has interrupts disabled and is set to generate a `dbexception` intercept. Macro `CORE_PROTECTED(x)` evaluates to true iff the core identified by `x` is switched to guest-mode.

leverages memory integrity to trap and intercept any changes to the LAPIC memory-mapped I/O page by the guest. More specifically, XMHF maps the LAPIC ICR to a page in its own memory $\mathcal{M}$ during initialization.

Subsequently, assuming $\varphi(\mathcal{M})$ holds, any writes to the LAPIC ICR by the guest causes the hardware to trigger a `npf` intercept. XMHFCORE handles the `npf` intercept, disables guest interrupts and sets the guest *trap-flag* and resumes the guest. This causes the hardware to immediately trigger a `dbexception` intercept, which is then handled by XMHFCORE to process the instruction that caused the write to the LAPIC ICR. If a SIPI command was being written to the ICR, XMHF voids the instruction and instead runs the target guest code on that core in *guest-mode*.

Thus, ensuring **MAC** (b) on multi-core CPUs 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) $ih\_$`dbexception`$()$ prevents write to LAPIC ICR on detecting a SIPI command and instead runs the target guest code on C in *guest-mode*.

The key insight is that this property can be verified by proving the validity of a properly inserted assertion in XMHF. Specifically, we engineer the verification as follows. Figure 6 shows the outline of the $ihub()$ function. Note the inserted assertion checks that the appropriate core is switched to guest-mode if the `dbexception` intercept is triggered. In our experiments, we use CBMC [19] to perform this model checking step. Our experiments and results are presented in detail in Section 4.6.

Note that the statements added to $ihub()$ are for verification purposes only. They are eliminated by the preprocessor while compiling the production version of XMHF, ensuring that no unnecessary statement is executed at runtime.

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

Figure 7: Outline of function `setprot()`, which is used to make all changes to the MMU and IOMMU tables.

4.4 Discharging Verification Conditions

VC1 is discharged by manually inspecting the `emhf_runtime_main` function of XMHF. Recall that this function is the $init()$ for XMHF. In particular, we manually verified that the function assigns each entry in the MMU table and the IOMMU table such that all addresses in $\mathcal{M}$ are designated read-only.

In the context of XMHF, discharging **VC2** reduces to verifying that the execution of $ihub()$ preserves $\varphi(\mathcal{M})$. We discharge this verification condition via software model checking. In particular, we verify that the execution of $ihub()$ does not modify permissions of any address in $\mathcal{M}$. Again, the key insight is that this property can be verified by proving the validity of a properly inserted assertion in XMHF. Specifically, we engineer the verification as follows:

1. The hypervisor memory is maintained in a contiguous set of addresses beginning at `HVLO` and ending at `HVHI`. This means that preserving $\varphi(\mathcal{M})$ reduces to ensuring that permissions of memory addresses between `HVLO` and `HVHI` are unaltered.
2. All changes to MMU and IOMMU tables are performed in a function called `setprot()`. Figure 7 shows the outline of `setprot()`. Note that every statement that potentially modifies the permission of a memory address a is preceded by an assertion that checks that $a <$ `HVLO` $\wedge\ a >$ `HVHI`.

Therefore, we discharge **VC2** by verifying that no execution of $ihub()$ leads to the failure of the assertion embedded in `setprot()`. Once again, we used CBMC to check the validity of this assertion. Our experiments and results are presented in detail in Section 4.6.

Again, note that the assertions inserted in `setprot()` are for verification purposes only. They are eliminated by the preprocessor while compiling the production version of XMHF, ensuring that there are no unnecessary assertion checks at runtime.

4.5 Development Compatible Verification

The mechanisms used to ensure properties **MAC** (a), **CFI**, **MOD**, and **ATOM** discussed in Section 4.2 are independent of XMHF's

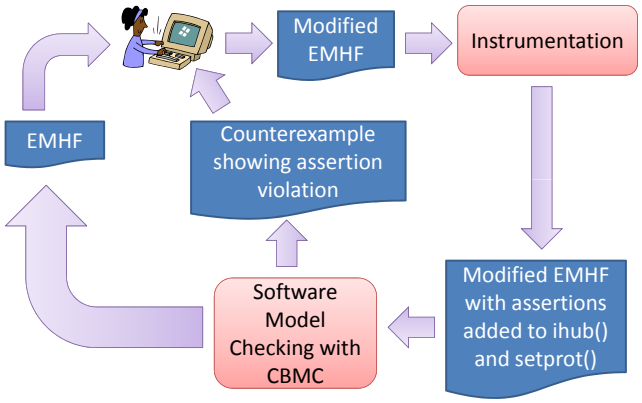

Figure 8: Overview of Development Compatible Verification for XMHF.

implementation. They depend either on the hardware or on design choices embedded in XMHF. These mechanisms do not change during XMHF's development, and are therefore development compatible. We now discuss how the verification of guest-mode execution – i.e., **MAC** (b) – and the discharging of verification conditions were both achieved in a development compatible manner for XMHF.

Development Compatible Verification of Guest-Mode Execution. Recall our approach to verify guest-mode execution presented in Section 4.3. Note that verification of $ihub()$ requires no user-supplied invariants or annotations. Therefore, after any change to XMHF's implementation, we are able to re-verify $ihub()$ with minimal manual effort. In particular, if there are no changes to $ihub()$, then we simply repeat the verification with CBMC. If there are changes to $ihub()$, we ensure that $ihub()$ is consistent with Figure 6 before re-verifying with CBMC. Hence, our approach to verify guest-mode execution is development compatible.

Development Compatible Discharging of Verification Conditions. Recall our approach to discharge **VC1** and **VC2** on XMHF presented in Section 4.4. Although **VC1** is discharged manually, it requires inspection of a small piece of code (approximately 400 SLOC), and is therefore development compatible. Also, the discharging of **VC2** requires no user-supplied invariants or annotations. Therefore, after any change to XMHF's implementation, we are able to re-discharge **VC2** with minimal manual effort. In particular, if there are no changes to `setprot()`, then we simply re-verify $ihub()$ using CBMC. If there are changes to `setprot()`, we ensure that `setprot()` is consistent with Figure 7 before re-verifying $ihub()$ with CBMC. Hence, our approach to discharge **VC2** is development compatible as well.

Figure 8 shows the overall development compatible verification applied to XMHF. The instrumentation was done manually but was trivial for the vast majority of modifications to XMHF since they did not alter $ihub()$ or `setprot()`.

4.6 Experimental Results

In this section, we report on our experiments to discharge the two verification conditions **VC1** and **VC2**, as well as to verify guest-mode execution in XMHF. As discussed earlier (see Sections 4.3 and 4.4), these verification problems were reduced to providing the validity of assertions in a sequential C program P. We now describe 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.

Experience with CBMC . CBMC [19] is a bounded model checker for verifying ANSI C programs. It supports advanced C features like overflow, pointers, and function pointers, and is therefore uniquely suited to verify system software such as XMHF. CBMC is only able to verify programs without unbounded loops. While XMHF contains such loops in general, all reachable loops in P are bounded. 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 had trouble handling 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 (the latest public release as of this writing). 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 deference unallocated memory. The remaining six buggy programs (P_1^L – P_6^L) have logical errors that cause assertion violations. In each case, CBMC 4.1 finds the errors successfully. Table 1 summarizes the overall results for CBMC 4.1. Note that the SAT instances produced are of non-trivial size, but are solved by the backend 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 time for parsing, transforming the program to an internal representation (called a GOTO program), slicing, simplification, and generating the SAT formula.

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 [41], SATABS [21], and WOLVERINE [47]. In each case, we used the latest publicly available version of the model checker. All these model checkers are able to verify programs with loops and use an approach called Counterexample Guided Abstraction Refinement (CEGAR) [13, 20] combined with predicate abstraction [35]. 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. Further details are presented in Table 1.

BLAST only accepts preprocessed C code. Therefore, to provide consistent input to all model checkers, we first preprocessed XMHF source code with `gcc`. This resulted in about 237 KLOC for each of our eleven target programs, even though the actual XMHF implementation is about 4700 LOC.

5. RELATED WORK

A sound architecture [15, 59] is known to be essential for the development of high quality software. Moreover, there has been a

Program	CBMC 4.1								SATABS 3.1		WOLVERINE 0.5c	
	OP	SP	VCC	Vars	CLS	DPT	Time	Mem	Iter	Mem	Iter	Time
P	1654	1452	111	437688	1560024	24.813	75.75	1958	56	594	1	168.47
P_1^M	1667	1465	116	438172	1561191	26.964	80.85	1959	56	594	1	168.52
P_2^M	1668	1466	116	438308	1561585	24.840	78.76	1959	56	594	1	168.60
P_3^M	1669	1467	116	438436	1561919	24.823	78.96	1910	56	594	1	297.17
P_4^M	1653	1451	117	463813	1668782	25.707	79.85	1910	56	594	1	247.28
P_1^L	1679	1477	111	476241	1728039	28.325	81.90	1910	56	594	1	168.64
P_2^L	1654	1452	111	437538	1559556	24.894	78.77	1910	56	594	1	205.72
P_3^L	1652	1450	111	437676	1559956	24.789	78.90	1910	56	594	1	296.91
P_4^L	1634	1441	111	437684	1560013	24.983	78.74	1910	56	594	1	173.81
P_5^L	1652	1450	111	437687	1560021	24.532	77.70	1910	56	594	1	281.44
P_6^L	1652	1450	111	437686	1560018	24.672	78.80	1958	56	594	1	275.40

Table 1: Summary of experimental results with CBMC, SATABS, and WOLVERINE. 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; DPT = time (sec) taken by SAT solver; Time = total time (sec); Mem = maximum memory (MB); in all cases, SATABS timed out and WOLVERINE ran out of memory; Iter = number of CEGAR iterations that were started before time or memory out.

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 [69]. 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 [26, 32] has been used to model and verify security properties of a number of different classes 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 with the hypervisor to a set of intercept handlers is crucial for the feasibility of DRIVE.

Control flow integrity (CFI) is one of the key system properties on which DRIVE relies. Techniques for ensuring CFI [8] have been widely studied and implemented. In essence, all source code analysis techniques assume CFI in some form. We therefore consider CFI to be an important but complementary problem.

There has been considerable work on verifying security at the level of models. For example, Guttman et al. [37] employ model checking to verify information-flow properties of SELinux. Lie et al. verify XOM [48] using MURPHI[3]. XOM is a hardware-based approach for tamper-resistance and copy-resistance. Mitchell et al. [51, 52] use MURPHI to verify the correctness of (and find bugs in) security protocol specifications. Franklin et al [29, 30] have developed a set of techniques for parametric verification of security properties of models of hypervisors. In contrast, we focus on verifying security at the level of source-code.

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 [18, 38, 71] target a general class of bugs. Others focus on specific types of errors, e.g., Kidd et al. [44] detect atomic set serializability violations, while Emmi et al. [27] verify correctness of reference counting implementation. All these approaches require abstraction, e.g., random isolation [44] or predicate abstraction [27], to handle source code, and therefore, are unsound and/or incomplete. In contrast, our focus is on a methodology to develop a hypervisor that achieves a specific security property against a well-defined attacker.

Finally, there has been several research projects on security of operating system and hypervisor implementations. Neumann et al. [53], Rushby [56], and Shapiro and Weber [58] propose verifying the design of secure systems by manually proving properties using a logic and without an explicit adversary model. A number of groups [40, 45, 68] have employed theorem proving to verify security properties of OS implementations. Barthe et al. [14] formalized an idealized model of a hypervisor in the Coq proof assistant and Alkassar et al. [9, 10] and Baumann et al. [16] annotated the C code of a hypervisor and utilized the VCC [22] 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 verification of the SEL4 operating system [45] required several man years effort. In contrast our approach is more automated but focuses on the specific property of memory integrity. Franklin's thesis [31] reports initial results on using CBMC to model check integrity properties of related small-TCB hypervisors' runtimes (300 LOC). This work demonstrates that CBMC can be used to check integrity properties with very little manual input for a much larger hypervisor runtime (4700 LOC).

6. DISCUSSION AND CONCLUSION

We presented an approach, called DRIVE, to design, develop and automatically verify integrity-protected hypervisors. We also validated DRIVE by using it to develop an extensible and modular hypervisor framework called XMHF [65]. In particular, the verification steps involved were performed on the actual source code of XMHF– consisting of about 4700 lines of C code – using the CBMC model checker. Our experience suggests that DRIVE is applicable to develop integrity-protected hypervisors of realistic complexity.

Of the software model checkers that we experimented with, CBMC was the only one that succeeded in the verification tasks posed by XMHF and DRIVE. We believe that this is due to a combination of two factors: (i) the reachable code in XMHF had only bounded loops – CBMC would fail if this was not the case; (ii) CBMC models

[3] `http://verify.stanford.edu/dill/murphi.html`

C programs at the bit-level and performs no abstraction. Other software model checkers that we tried kept iterating their abstraction-refinement loops till they ran out of time or memory.

We believe that several factors contribute to enable automated verification with DRIVE, particularly in the case of XMHF. First, even though a virtualized system is inherently concurrent, the verification conditions of DRIVE are discharged by analyzing only sequential programs. Verification of sequential programs is understood to be more tractable. Second, XMHF is implemented in C, and several model checkers for sequential C programs are publicly available. Finally, we were able to engineer the verification process to be development compatible – it was repeated automatically and routinely as part of XMHF's build process.

The system properties and verification conditions required by DRIVE are sufficient to guarantee memory integrity. However, we have not proven them to be necessary. Nevertheless, they serve as a useful checklist for reasoning about the integrity of other hypervisors as well. If a certain hypervisor satisfies the DRIVE properties and verification conditions, then, by Theorem 1, it is also integrity protected. If, on the other hand, it violates a certain property or verification condition, then the failure offers a starting point to detect integrity-related vulnerabilities.

We believe that DRIVE provides a good starting point for research and development on hypervisors with rigorous and "designed-in" security guarantees. One direction for future work is to extend DRIVE to other security properties, such as confidentiality. An immediate challenge here is that such a property may not be as easily expressible as integrity since the attacker's interface is much less well-defined due to the possibility of covert channels etc. Yet another direction is to develop a hypervisor that supports multiple guests. The question here is whether DRIVE still guarantees integrity in such situations, and if not, how it must be updated so that integrity is assured.

7. REFERENCES

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

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

[3] The CPU hardware emulation does not properly handle the Trap flag. CVE-2008-4915 (under review), 2008.

[4] Directory traversal vulnerability in the shared folders feature. CVE-2008-0923 (under review), 2008.

[5] Heap-based buffer overflow in Xen 3.3, when compiled with the XSM:FLASK module, allows unprivileged domain users (domU) to execute arbitrary code via the flaskop hypercall. CVE-2008-3687, 2008.

[6] Multiple buffer overflows in openwsman allow remote attackers to execute arbitrary code. CVE-2008-2234, 2008.

[7] VMware patches for ESX and ESXi resolve a critical security vulnerability. VMSA-2009-0006, 2009.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity principles, implementations and applications. *ACM Transactions on Information and System Security*, 13(1), 2009.

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

[10] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated Verification of a Small Hypervisor. In *Proc. of VSTTE*, volume 6217, 2010.

[11] AMD Inc. Amd64 architecture programmer's manual volume 2: System programming. 2011.

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

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

[14] 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.

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

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

[17] D. Bell and L. La Padula. Secure Computer Systems: A Refinement of the Mathematical Model. Technical Report MTR-2547, Vol 3, MITRE Corp., 1974.

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

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

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5), 2003.

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

[22] 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.

[23] F. J. Corbato and C. T. Clingen. A managerial view of the multics system development. In *Conference on Research Directions in Software Technology*, 1977.

[24] F. J. Corbato, J. H. Saltzer, and C. T. Clingen. Multics – the first seven years. In *Spring Joint Computer Conference*, 1972.

[25] S. Crosby and D. Brown. The virtualization reality. *ACM Queue*, 4(10), 2006.

[26] A. Datta, J. Franklin, D. Garg, and D. K. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *Proc. of IEEE S&P*, 2009.

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

[28] L. Fraim. SCOMP: A solution to the multilevel security problem. In *IEEE Computer*, 1983.

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

[30] 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.

[31] J. D. Franklin. Abstractions for model checking system security. Technical report CMU-CS-12-113, School of Computer Science, Carnegie Mellon University, 2012. http://reports-archive.adm.cs.cmu.edu/anon/2012/CMU-CS-

12-113.pdf.

[32] D. Garg, J. Franklin, D. K. Kaynar, and A. Datta. Compositional System Security with Interface-Confined Adversaries. *Electronic Notes in Theoretical Computer Science*, 265, 2010.

[33] V. Gligor, C. Burch, R. Chandersekaran, L. Chanpman, M. Hecht, W. Jiang, G. Luckenbaugh, and N. Vasudevan. On the Design and the Implementation of Secure Xenix Workstations. In *Proc. of IEEE S&P*, 1986.

[34] V. D. Gligor. Design and implementation of secure XENIX. *IEEE Transactions on Software Engineering*, 13(2), 1987.

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

[36] T. C. Group. Trusted platform module main specification, Version 1.2, Revision 103, 2007.

[37] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1), 2005.

[38] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. of PLDI*, volume 37(5), 2002.

[39] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *IEEE CollaborateCom*, 2005.

[40] 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.

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

[42] Intel Corporation. Intel trusted execution technology – software development guide. Document number 315168-005, June 2008.

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

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

[45] 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.

[46] K. Kortchinsky. Cloudburst: A VMware guest to host escape story. Black Hat, 2009.

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

[48] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proc. of IEEE S&P*, 2003.

[49] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Trans. Software Eng.*, 37(3), 2011.

[50] 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*, 2010.

[51] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proc. of IEEE S&P*, 1997.

[52] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Proceedings of the Seventh USENIX Security Symposium*, pages 201–216, 1998.

[53] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, SRI International, 1980.

[54] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Comm. ACM*, 17, 1974.

[55] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2007.

[56] J. M. Rushby. Design and Verification of Secure Systems. In *Proc. of SOSP*, 1981.

[57] 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.

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

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

[60] R. E. Smith. Cost profile of a highly assured, secret generating system. 4(1), 2001.

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

[62] U.S. Department of Defense. Trusted computer systems evaluation criteria. (Orange Book) CSC-STD-001-83, DoD Computer Security Center, 1983.

[63] U.S. Department of Defense. Trusted computer systems evaluation criteria. (Orange Book) 5200.28-STD, National Computer Security Center, 1985.

[64] U.S. Department of Defense. Rating maintenance phase program document version 2. Technical Report NCSC-TG-013-95, National Computer Security Center, 1995.

[65] A. Vasudevan, J. M. McCune, and J. Newsome. "It's an app. It's a hypervisor. It's a hypapp.": Design and Implementation of an eXtensible and Modular Hypervisor Framework. Technical Report CMU-CyLab-12-014, CMU CyLab, 2012.

[66] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: A Safe and Practical Environment for Security Applications. Technical Report CMU-CyLab-09-011, CMU CyLab, 2009.

[67] A. Vasudevan, N. Qu, and A. Perrig. XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms. In *Proc. of HICSS*, 2011.

[68] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM (CACM)*, 23(2), 1980.

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

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

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

[72] 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.