Monitoring Integrity Using Limited Local Memory

Yuki Kinebuchi, Shakeel Butt, Vinod Ganapathy, Liviu Iftode, and Tatsuo Nakajima

***Abstract*—System integrity monitors, such as rootkit detectors, rely critically on the ability to fetch and inspect pages containing code and data of a target system under study. To avoid being infected by malicious or compromised targets, state-of-the-art system integrity monitors rely on virtualization technology to set up a tamper-proof execution environment. Consequently, the virtualization infrastructure is part of the trusted computing base. However, modern virtual machine monitors are complex entities, with large code bases that are difficult to verify. In this paper, we present a new machine architecture called limited local memory (LLM), which we use to set up an alternative tamper-proof execution environment for system integrity monitors. This architecture builds upon recent trends in multicore chip design to equip each processing core with access to a small, private memory area. We show that the features of the LLM architecture, combined with a novel secure paging mechanism, suffice to bootstrap a tamper-proof execution environment without support for hardware virtualization. We demonstrate the utility of this architecture by building a rootkit detector that leverages the key features of LLM. This rootkit detector can safely inspect a target operating system without itself becoming the victim of infection.**

***Index Terms*—Local memory, multicore, system integrity.**

I. Introduction

In recent years, there has been extensive research on applying virtual machine technology to problems in security. This research has been fueled both by the wide availability of virtualization, such as in the cloud infrastructure, and the attractive security guarantees provided by virtual machine monitors (VMMs). VMMs implement a software layer that virtualizes system resources (the *hypervisor*) so that the operation of one virtual machine does not affect the resources used by another. This feature allows a security monitor to be easily isolated from the system under study (the *target*), in order to remain tamper-proof and function effectively. Such isolation is central to the architecture of system integrity monitors that inspect the code and data of a potentially compromised target. For instance, *rootkit detectors* (*e.g.,* [39], [56], [41], [9], [40], [5], [21], [26], [20], [9],[34], [47]) must be able to monitor a target operating system for malicious changes that affect the integrity of its code and data without exposing themselves to attack. Contemporary techniques to achieve isolation use VMMs to execute the rootkit detector and the target operating system within different virtual machines (VMs). Hence, VMMs must be part of the trusted computing base (TCB).

However, VMMs represent a *software-centric solution* to the problem of isolation. As with any other software layer, they are also prone to the pitfalls of the software development process. Modern VMMs contain thousands of lines of code and exploitable vulnerabilities are routinely reported in them [15], [16], [18], [17], [3], [27], [44]. For instance, in the Xen VMM (v4.1), the hardware virtualizing layer (*i.e.,* the Xen hypervisor) alone accounts for approximately 150 K lines of code. In addition, a privileged VM that is used for administrative purposes (*i.e.,* dom0) runs a software stack with a full-fledged operating system, including drivers for virtual devices and user-level utilities, which constitute several million lines of code. The TCB of the Xen VMM includes both the hypervisor and the dom0 VM. It is no longer reasonable to assume that such commodity VMMs can be easily verified, which is one of the requirements for an entity that constitutes the TCB [4]. A compromised VMM completely subverts the security of the system, exposing the virtual machines on that platform to a variety of threats (*e.g.,* see [31], [28]).

The primary reason behind the large code base in VMMs is that virtualization is used to solve multiple problems: better resource utilization, fault isolation, better management, safe sharing of resources, replaying, debugging, and providing isolation to security tools. More recently, virtualization has become the driving force behind the multimillion cloud industry. More vendors are joining the wagon of cloud business and are competing for more features and functionalities. These trends together have increased the size of the TCB in general purpose VMM solutions, such as Xen, KVM, VMWare and HyperV.

The growing complexity of VMMs has motivated researchers to consider solutions that reduce the amount of code in the TCB. Some researchers have focused on developing tailored VMMs for specific tasks (instead of aiming to be general purpose) in an effort to reduce the size of the TCB (*e.g.,* TrustVisor [36], SecVisor [47] and BitVisor [48]). Researchers have also tried to rearchitect the design of VMMs [50] and disaggregate commodity VMMs to implement privilege separation [37], [11] therby reducing the size of the TCB. The net effect of these efforts is that the TCB size reduces to the order of a few thousand lines of code. Nevertheless, this TCB still has to

Manuscript received March 08, 2012; revised July 18, 2012, October 25, 2012, and January 25, 2013; accepted May 18, 2013. Date of publication June 04, 2013; date of current version June 21, 2013. This work was supported in part by the National Science Foundation under Grants CNS-0831268, CNS-0915394, and CNS-1117711. Early versions of this paper were presented at the 16th Pacific Rim International Symposium on Dependable Computing, December 2010, and the IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications, August 2011. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miodrag Potkonjak.

Y. Kinebuchi was with the Distributed and Ubiquitous Computing Laboratory, Department of Computer Science and Engineering, Waseda University, 3-4-1 Ohkubo, Shinjuku-ku, Tokyo, Japan. He is now with Nvidia Japan ATT New Tower, Tokyo 107-0052, Japan (e-mail: yukikine@gmail.com).

S. Butt, V. Ganapathy, and L. Iftode are with the Department of Computer Science, Rutgers University, Piscataway, NJ 08854 USA. (e-mail: shakeelb@cs.rutgers.edu; vinodg@cs.rutgers.edu; iftode@cs.rutgers.edu).

T. Nakajima is with the Distributed and Ubiquitous Computing Laboratory, Department of Computer Science and Engineering, Waseda University, 3-4-1 Ohkubo, Shinjuku-ku, Tokyo, Japan (e-mail: tatsuo@dcl.info.waseda.ac.jp).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2013.2266095

interact with untrusted code to handle VM-Exits, hypercalls or emulate privileged instructions. A number of vulnerability reports in commodity VMMs [15], [16], [18], [17], [3], [27], [44] have demonstrated that even these interactions can be exploited maliciously.

More recently, researchers have proposed a *hardware-centric approach*, NoHype [28], [29], to provide a cloud computing environment that leverages emerging hardware features to isolate "virtual machines" running on the same physical machine. While NoHype provides several attractive security benefits, such as the ability to isolate mutually untrusted virtual machines from each other, *it does not allow virtual machines to inspect the memory of each other*, a feature that is necessary for the implementation of security monitors such as rootkit detectors. In this paper, our focus is on *hardware-centric solutions* to the problem of isolating security tools from monitored vulnerable VMs.

We developed a hardware-centric approach to establish a tamper-proof environment for security monitors without using virtualization. Our main contribution is in showing that a modest amount of hardware support allows security monitors to securely inspect the state of a target system. In a manner akin to VMMs, our approach isolates security monitors from the target system, which may be compromised or malicious. However, it does so (1) by making minor enhancements to a multicore hardware with core local memory; and (2) by executing the code in TCB without direct interaction with the monitored system (*e.g.,* via VM-Exits).

We propose and study a novel hardware architecture, called *limited local memory* (LLM), that isolates security monitors from target operating systems, while also providing monitors the ability to inspect the target's state. LLM is inspired by the recent efforts of the major multicore vendors [23], [22], [38] to equip each processing core with local storage in addition to shared main memory. We show that the LLM architecture can successfully play the isolation functionality of the VMM by enabling tamper-proof execution of system integrity monitors.

An LLM-based machine extends a standard multicore machine by adding local memory to each core [23], [22], [38] and by allowing one core to become privileged (see Fig. 1). In the spirit of virtualization, we refer to this privileged core as *core0* and every other core as a *coreU*. Core0 has the ability to freeze the execution of coreU processors, returning control to itself. While coreUs can access each other's local memory, core0 has exclusive access to its local memory.

We show that these features enable LLM to set up a tamper-proof execution environment for system integrity monitors. We illustrate this fact by implementing a rootkit detector on an LLM machine. The rootkit detector itself executes on core0 (which runs its own kernel, and is part of the TCB) to monitor the execution of the operating system executing on coreUs (*i.e.,* the target operating system). The rootkit detector shares the main memory with the target kernel, and is therefore able to inspect it for the presence of rootkits.

Despite sharing main memory with the target, we show that an LLM-based rootkit detector can remain untampered even in the presence of rootkits that infect the target. We achieve this goal using a novel ***secure paging mechanism***, which ensures that: (1) all code execution and data access on core0 (which runs the rootkit detector) happens only from its local memory; and (2) all code and data pages stored in shared main memory are first authenticated before they are paged into core0's local memory. These two properties allow the secure paging mechanism to prevent attacks directed at the rootkit detector itself.

LLM reduces the size of TCB to around 10KLOC, which is comparable to TCB reduced virtualization based solutions, specialized hypervisors (*e.g.,* TrustVisor [36], SecVisor [47] and BitVisor [48]) and minimal hypervisor (*e.g.,* Nova [50]). The major advantage of LLM over these virtualization-based systems is that the code that runs as TCB has no direct interaction with the monitored operating system. In the virtualization based systems, the TCB code has to handle directly many events from the monitored operating system like VM-Exits, hypercalls and emulation of privileged instructions, hence, they have a narrow window of possible vulnerabilities.

In summary, the main contributions of this paper are:

- *LLM architecture and secure paging.* We present the limited local memory (LLM) architecture and describe its key features. We demonstrate that LLM, when combined with our secure paging mechanism, can enable tamper-proof execution of system integrity monitors. The key advantage of the LLM architecture is that it enables secure execution of system integrity monitors without support for hardware virtualization.
- *Rootkit detection using LLM.* We demonstrate the utility of the LLM architecture by building a rootkit detector that leverages the key features of LLM. We show that the detector can safely inspect code and data of a potentially compromised operating system without itself being infected by rootkits. We also show that the rootkit detector can operate in parallel with the target, thereby enabling low-overhead detection.

II. The LLM Architecture

The two key features of the LLM architecture are that: (1) each processor core is equipped with local memory, and (2) one processor core can be made privileged. In this section, we discuss each of these features, and prior work that inspired these features in LLM.

The use of local memory in LLM architecture is inspired by prior work on on-chip local memory [43], [7], [42], [8], [52], [38], [23], [22]. On-chip local memory has previously been proposed as a mechanism to provide predictable program behavior, especially in real-time systems. Their use is primarily motivated by the increasing gap in the speed of processors and main memory. While the use of caches can reduce this gap, the adaptive and dynamic behavior of caches can introduce unpredictable program behavior, making them hard to use in real-time systems. Consequently, computer architects have explored the use of local memory to address this problem. Commercial processors equipped with such local memory include ColdFire MCF5249, PowerPC 440, MPC5554, ARM 940, ARM 946E-2 and AT91M40400 [43], [7]. The most popular models of local memory proposed in the literature fall into two categories: *cache locking* [42], [8], [52] and *scratchpad* [7]. In cache-lock-based local memory, hardware support is leveraged to control the contents of the cache. Data that must be cache-resident is loaded

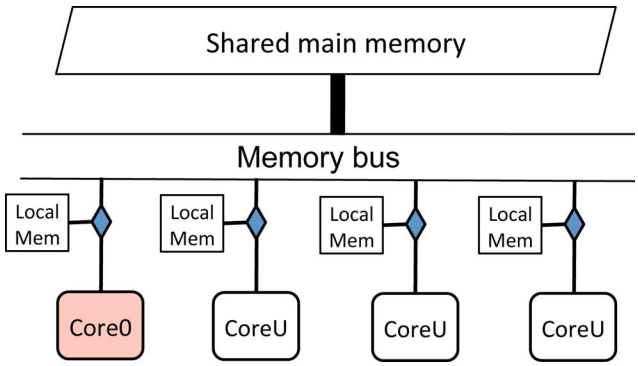


Fig. 1. LLM extends a standard multicore machine by adding local memory to each core. One core, called core0, is privileged, in that it can halt the execution of the other cores, which are also called coreUs. Core0's execution cannot be halted by coreUs. Each coreU can access the local memory of other coreUs, possibly with additional latency. Core0's local memory cannot be accessed from coreUs.

into the cache, and the cache replacement policy is disabled. In contrast, in scratchpad local memory, software has fine-grained control over the contents of local memory, which is addressable through the physical address space. Puaut and Pais [43] provide a good comparison of these two techniques in different scenarios. LLM's local memory model is similar to the proposals on scratchpad local memory.

More recently, with the advent of multicore systems and the rapid increase in the number of cores per chip, researchers have begun to face scalability problems in implementing cache coherence techniques. Consequently, computer architects are once again exploring the design space of per-core local memory to address this scalability problem. For example, in the RP1 processor announced by Hitachi [38], each core is equipped with 152 KB of local memory. The Intel's single-chip cloud computer (SCC) [22], [23], a research multicore processor, also equips each core with its own private off-chip DRAM [24, Page 52]. The difference between Hitachi's RP1 and Intel's SCC is that, in RP1, each core can access local memory of other cores with additional latency, making it akin to NUMA architecture, while in SCC, the local memory of each core is private. LLM's local memory architecture is directly inspired by both Hitachi's RP1 and Intel's SCC.

In addition to local memory, LLM designates one processor core, called core0, to be privileged. Core0's privilege is twofold. First, *core0 can reset or halt other processors*. Second, *core0's local memory is not accessible to other cores*.

In LLM, core0 is selected at boot time among all available cores across all multicore processors in the system using an extended bootstrap processor (BSP) selection algorithm [25]. In a traditional multicore multiprocessor system, the bootstrap processor (BSP) selection algorithm is executed to select the first core (or processor) to initialize the system. After initializing the system and operating system, the BSP signals to other cores (or processors) through inter processor interrupts (IPIs), to start execution. In the LLM architecture, we have incorporated the selection of core0 with the BSP selection process. The core selected by the BSP algorithm is designated as core0 by setting a hardware bit associated with that core. Once this bit is set, only core0 has the privilege to clear it and relinquish its privilege. The local memory of core0 is protected from access by other cores. Core0 is responsible for loading the target operating system on the coreU processors.

As we will describe in Section III, these features of core0 together with our secure paging mechanism allow the LLM architecture to bootstrap a tamper-proof execution environment for system integrity monitors. Intuitively, core0 acts as the hardware root of trust: first it acquires control of the machine during the boot time, then loads the secure paging mechanism into its limited local memory area before initializing the rest of the system. The monitor is entirely executed by core0 from its local memory. A secure paging mechanism allows core0 to check the integrity of the monitor each time its pages are loaded from shared main memory into its local memory. As other cores cannot access core0's local memory, they cannot tamper with the execution of the integrity monitor, which can then securely oversee the integrity of code and data accessed by the coreUs. If the monitor detects an integrity violation, either to its own code or data or to that of the target, it can instruct core0 to halt the execution of the coreUs and raise an alert.

The design of LLM closely resembles existing multicore architectures (*e.g.,* the Hitachi RP1 and Intel's SSC). We did so in the interest of deployability — as proposed in this paper, LLM can be implemented with only minor changes to these existing architectures. We made a conscious design choice of avoiding more complex design features that could potentially add utility to security monitoring at the cost of increasing the complexity of the underlying micro-architecture. For example, in LLM, core0 cannot access the internal register state of coreU. While this limits the ability of LLM to detect certain classes of attacks (see Section IV), we believe that the simpler resulting design of LLM lends itself to easy deployability.

III. Integrity Monitors Using LLM

In this section, we present the design of an integrity monitor that leverages the key features of the LLM architecture. We begin by stating the problem, defining the threat model and identifying the trusted computing base (TCB). Throughout, we focus on the *mechanisms* used by an LLM-enabled integrity monitor, which we consider the core contribution of this paper, and not on the *policies* used to detect integrity violations.

A. Goal

The goal of an integrity monitor is to oversee the execution of a *target* by inspecting its code and data. For this paper, we will assume that the target is an operating system whose code and data may be compromised by malicious software, such as rootkits. We only focus on mechanisms to protect operating system integrity because integrity monitors for user-space applications can be bootstrapped using an integrity-protected operating system. Moreover, the design of user-space integrity monitors is substantially similar to the operating system integrity monitors that we describe in this paper. The integrity monitor must be able to inspect the target for malicious software without itself becoming a victim of compromise. If it detects that the target has been compromised, it must be able to halt the execution of the target and take appropriate action, *e.g.,* report an alert to the end-user or an audit log. We do not consider the goal of recovering the target from compromise.

B. Machine Setup

To monitor the integrity of a target operating system, we require an LLM machine to be set up as follows:

- the integrity monitor executes on core0. The monitor's execution environment includes an operating system that controls core0 (henceforth called the *monitor operating system*) and a user-space program that encodes the target's integrity policy that must be applied to its code and data.
- the target operating system executes on coreUs.

This setup requires the LLM machine to execute two operating systems: a monitor operating system executing on core0, and the target executing on coreUs. In contrast, in a typical multicore machine, all cores execute the same operating system image. To enable this setup, core0 is first initialized during boot time, when the monitor operating system and the integrity monitor are loaded. Subsequently, the monitor operating system initializes the coreUs and loads the target operating system on these cores. The sequence is reversed during shutdown, when core0 terminates the target and halts the cores before itself halting execution. We defer the details of this setup to Sections III-E and V.

C. Threat Model and TCB

Having described the goal and the machine setup, we can now define the threat model and the TCB. We assume that the target operating system is vulnerable to attack and that its code and data may be compromised in malicious ways. Although the security monitor executes code and accesses data from core0's local memory, which is not accessible to the target, due to the limited size of the local memory, the shared main memory may have to be used for paging. When residing in the shared memory, these pages of the monitor are accessible to and may be maliciously modified by the target. The monitor must therefore be able to detect the attacker's attempts to modify its own code and data *i.e.,* it must have the ability to inspect its own code and data and halt execution in case integrity is compromised (it does so using secure paging).

In detecting these threats, we trust the following entities, which constitute our TCB:

- *the LLM hardware platform.* We trust that the hardware is implemented correctly in that: (1) core0 has the ability to halt the execution of the coreUs, while coreUs cannot reboot core0. A consequence is that core0 should first acquire control when the system is booted; and (2) the local memory of core0 is private (cannot be accessed from coreUs).
- *the BIOS and bootloader.* We trust the BIOS and bootloader to correctly load the secure pager and its associated data structures into core0's local memory at startup, where they will reside until machine shutdown.
- *the monitor operating system and its user processes.* We trust that the operating system and user processes executing on core0 are themselves not malicious. Because core0 may use the shared main memory as a backing store for its pages, the monitor operating system and its user processes may themselves be infected while paged out. We do allow such attacks within our threat model and detect such attacks using the secure paging mechanism; we only require that the monitor operating system and its processes be clean at boot time.

To provide the security assurances that we discuss in Section IV, the above components, which constitute the TCB, must function correctly and without compromise for the lifetime of the system.

An attacker can violate our assumption that the monitor operating system and its processes are trusted by directly modifying pages on disk before the machine is booted (so that a compromised operating system is loaded at boot time). We assume that such attacks can be detected using trusted computing technology (*e.g.,* trusted boot and attesting integrity of code at boot time using a TPM). Alternatively, the machine can be securely booted over the network or using a LiveCD containing an uncompromised operating system. We do not discuss such attacks in this paper in further detail.

D. Bootstrap Process

The monitor starts operation when the machine is first booted and continues to execute until the machine is shut down. During boot up, the bootloader loads the secure paging mechanism and initializes it in core0's local memory. The secure pager bootstraps a tamper-proof environment for the monitor operating system and its user processes. Secure paging is described in further detail in Section III-E. In addition to installing the secure pager, the bootloader also copies into core0's local memory several other data pages that remain resident there for the lifetime of the system (we describe this in detail in Section III-E). The rest of the code and data of the monitor operating system are loaded into the shared main memory. Then, the monitor operating system initializes its user processes that will check the integrity of the target. At this point, the monitor is initialized and can load the target operating system on the remaining cores.

The procedure of booting the target operating system on coreUs is similar to booting it on a traditional multicore machine, except that the target is initialized on the coreUs by the monitor rather than by the bootloader. In order to boot on an LLM machine, a minor change is required to commodity operating systems. Specifically, it must be modified to avoid allocating its data structures in the portion of the shared main memory that stores the monitor operating system's code and data. The purpose of this change is to prevent the monitor from raising an alert when an uncompromised target inadvertently uses pages utilized by the monitor operating system. However, in case rootkits that compromise the target modify this memory region, they will be detected by the monitor's secure paging mechanism.

In summary, the bootstrap process proceeds as follows:

(1) On system boot, core0 is selected, and starts executing the BIOS.
(2) The BIOS loads bootloader into memory and starts executing the bootloader.
(3) The bootloader loads the secure pager part of monitor OS into core0's local memory and the remainder of the monitor OS into shared memory. Then, it starts executing monitor OS, whose entry point must be in local memory.

(4) The monitor OS sets up the page table to allow code and data pages to be accessed only from local memory.
(5) The monitor OS loads the target operating system in the shared memory, and signals the coreUs to start executing it.

E. Monitor Operation

We now discuss the operation of the monitor during two different scenarios: (1) during normal operation, when it monitors an uncompromised target operating system; and (2) during attack, when it detects either that its own code and data pages have been corrupted, or that the target's integrity policy has been violated.

1) Normal Execution and Secure Paging: Once startup is complete, the monitor oversees the execution of the target. Because the monitor and the target share the main memory, the monitor is vulnerable to attacks from a compromised target. To avoid such attacks, the monitor's execution environment is set up to satisfy the following three invariants:

- [LocalCode]—*Ensures local code execution.* A code page can be executed by core0 only from its local memory. In case the monitor's code to be executed resides in the shared main memory, as described earlier, it must first be brought into core0's local memory.
- [LocalData]—*Ensures local data access.* A data page belonging to the monitor must first be brought into core0 local memory before it can be accessed
- [AuthLoad]—*Verifies integrity before loading.* The integrity of code and data pages belonging to the monitor must first be verified before they are brought into core0 local memory.

The enforcement of all three invariants is the responsibility of the *secure paging mechanism*, which is implemented within the monitor operating system. During startup, the bootloader loads the code of the secure pager into core0's local memory area. The secure pager also contains precomputed hash values for the monitor's code and read-only data pages (a whitelist). The memory pages containing the code of the secure pager and these hash values remain resident in core0's local memory for the lifetime of the system, *i.e.,* they are never paged out into main memory or to disk.

The monitor operating system revokes permissions to execute, read or write code and data pages of the monitor when they reside in the shared main memory by suitably setting permission bits in the monitor operating system's page table entries. Therefore, any attempt to access these pages results in a fault that is handled by the secure pager. The secure pager handles this page fault by bringing the page from the shared main memory into the local memory only after computing the hash of the page and checking it against the whitelist. If there is a match, the page is loaded into local memory and the corresponding access is turned on. If there is no match, the secure pager triggers an alert.

Because core0 local memory has limited size, pages may need to be evicted as the monitor operates. Before eviction, the secure pager must compute the hash of the selected page and store it in the local memory. Then, the secure pager revokes the access permissions for this page so that further accesses from core0 will trigger a page fault. In the absence of malicious target that may modify the monitor code, the monitor page does not change while residing in shared main memory, and hence, the hash values will match next time the secure pager attempts to load this page into the local memory as a result of a legitimate core0 access. The secure pager's authenticated load facility (computing and comparing hashes) is inspired by the techniques implemented in prior work on Patagonix [34] and SecVisor [47]. Fig. 3 summarizes the secure paging mechanism.

In order to enable secure paging, several code and data pages must remain resident in core0's local memory for the lifetime of the system:

- the code of the monitor operating system's secure paging mechanism. This includes code responsible for hashing pages loaded from shared main memory, comparing these hashes against saved values, and terminating execution of coreUs if an integrity violation is detected;
- hash values of the code and data of the monitor operating system and its user processes. Some of these hash values are precomputed (*e.g.,* those for code and static data pages) while others are computed dynamically by the secure paging mechanism itself when performing pageouts.
- the monitor operating system's data structures that are *directly* accessed by the hardware. These include data pages that store its page tables and interrupt vector tables. Typically, only an active root page table, and second and third-level page tables need to be resident in local memory; any other page tables that are not directly accessed by hardware can be swapped out to shared main memory and can themselves be verified by the secure paging mechanism.

2) Execution Under Attack: During normal operation, the secure paging mechanism checks the integrity of the monitor's code and data (for both the monitor operating system and its user processes). The target can define its own integrity policies, which are encoded within and enforced by the monitor's user processes by inspecting the target's code and data pages stored in shared memory.

The monitor raises an alert under one of these conditions:

(1) *the monitor's integrity is violated.* The secure paging mechanism can detect when a compromised target violated the integrity of the monitor's code and data.
(2) *the target's integrity policies are violated.* A rootkit may compromise the target operating system (*e.g.,* by corrupting the system call table or function pointers within the kernel), thereby violating code or data integrity. If supplied with a suitable policy, the monitor process that checks the target's memory pages will detect this integrity violation.

In each of these cases, core0 issues an interprocessor interrupt that will halt the execution of the coreUs, returning control to itself. The target will therefore be halted, and will be unable to make further changes to shared memory. The monitor operating system will then acquire control of a peripheral device, such as the console, a serial port, or the hard disk, to emit diagnostic information, which may include a warning on the console, as well as a snapshot of shared main memory (for forensic purposes).

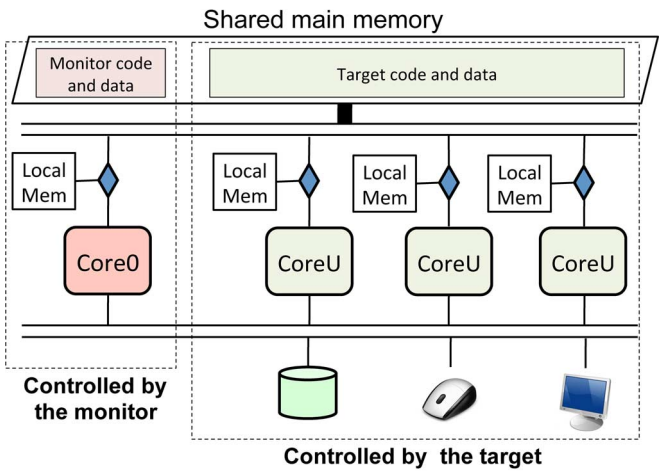


Fig. 2. Configuration of an LLM machine just after startup is complete and during normal operation of the system. Core0 executes the monitor operating system and its processes from its local memory. The target executes on coreUs and controls the peripheral devices. Core0 acquires control over peripherals only when it detects an integrity violation. Main memory is shared by the monitor and target operating systems.

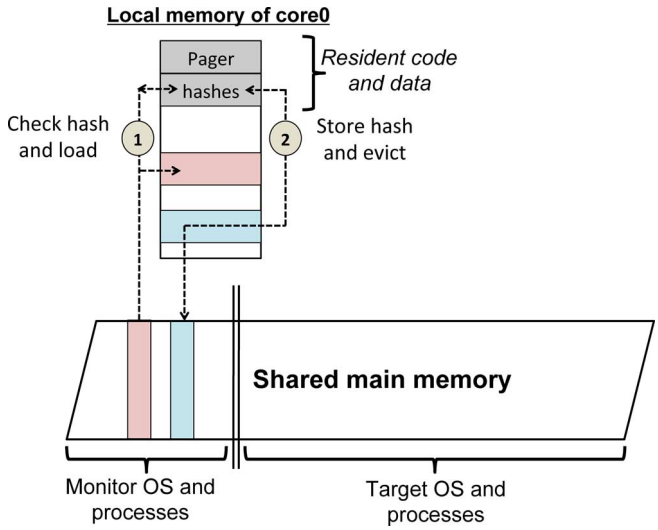


Fig. 3. Secure paging mechanism. The pager is loaded into core0's local memory by the bootloader during startup. The code of the pager and the pages containing a whitelist of hashes remain resident in local memory until the machine is shut down. (1) When core0 needs to access a page that is not available in local memory, it results in a page fault, which is handled by the secure pager. The secure pager hashes the corresponding shared memory page, checks the hash against stored values, and loads the page if it has not been modified. (2) When a page must be evicted from local memory, the secure pager computes its hash before paging it out to the shared main memory.

At this point, the end-user can take appropriate actions, which may include restarting the machine or cleaning up the infection. We leave exploration of postcompromise user actions for future work.

F. Device Control

In our prototype system, all devices are controlled by the target operating system; this includes the disk, monitor and all input devices. The monitor operating system does not control any device during the course of normal operation. When the monitor detects that the target's integrity has been violated, it freezes the execution of coreUs and acquires control of an output device (*e.g.,* the screen or a serial port) to notify the end-user about the violation. During startup, the bootloader loads all the code and data of the monitor operating system and its user applications into shared memory, thereby obviating the need to access persistent storage devices over the course of normal operation. To achieve this goal, the monitor operating system must offer minimal functionality; in our implementation, we use xv6 [13] as the monitor operating system. Fig. 2 depicts the configuration of an LLM machine after startup is complete.

IV. Security Analysis

We analyze the security of an LLM-based integrity monitor based on: (1) its ability to protect itself from integrity violations; and (2) its ability to protect the target.

A. Ability to Protect Itself

The ability of an LLM-based integrity monitor to protect its own integrity from an infected target is predicated on the LocalCode, LocalData and AuthLoad invariants. We first discuss how enforcing these invariants ensures the integrity of the monitor and then analyze the enforcement of the invariants.

The invariants LocalCode and LocalData ensure that the code and data accessed by core0 (both the monitor operating system and its user processes) are not visible to the target, and hence, not visible to rootkits that may infect the target. Moreover, the invariant AuthLoad ensures that the monitor only executes approved code, as determined by a whitelist of hashes that is resident in core0's local memory. AuthLoad also ensures that the hash of a code or data page evicted from local memory is computed during eviction and checked when the page is loaded again into local memory.

The invariants themselves are enforced by the secure pager, which is never paged out of core0's local memory, and hence, cannot be accessed or modified by the untrusted target. Therefore, the security of the system is bootstrapped at startup when the bootloader initializes core0. At this time, it loads into core0's local memory pages containing the secure pager and a precomputed whitelist with the hashes of the monitor's code and static data pages. These pages reside in core0 local memory until the machine is shut down. The privilege granted to core0 means that its execution cannot be halted or altered by coreUs, which execute the target. Core0 does not interact with peripheral devices, which are controlled by the target until it detects an integrity violation. In this case, core0 halts coreUs before acquiring control over an output device to emit diagnostics.

Consequently, LocalCode, LocalData and AuthLoad are enforced for the lifetime of the system, together ensuring that core0 only executes approved code and reads approved data, which guarantees a tamper-proof execution environment for the monitor.

While the three invariants protect the *integrity* of the monitor from an infected target, its *confidentiality* may still be compromised by the target. For example, if the monitor uses a secret key to securely transmit diagnostic information outside the system and stores this key in shared main memory, it would be visible to the target. Although we have not attempted to protect the confidentiality of the monitor's code and data pages in our prototype

implementation, this goal can be achieved using a simple modification to the secure paging mechanism. When the secure pager is first loaded into core0's local memory, it could generate a secret key that also resides in core0's local memory for the lifetime of the system. The secure pager could be modified to use this key to encrypt the monitor's code and data pages when they reside in shared main memory. When core0 attempts to access an encrypted code or data page, the secure pager brings it into local memory, decrypts it, and checks its integrity as before. Since the monitor's pages are stored encrypted when in main memory, this modification also protects its confidentiality.

Finally, a malicious target may also attempt to inflict *availability* attacks on the monitor. For instance, if the machine is equipped with a programmable memory controller, the target could maliciously slow down or deny the secure pager from accessing the monitor's pages in shared main memory. The monitor could detect such availability attacks using timeouts, and pause the target operating system if it suspects an attack. However, we have not implemented any defenses against such attacks in the monitor of our prototype.

B. Ability to Protect the Target

An LLM-based integrity monitor oversees target execution to enforce target-specified integrity policies. Its ability to protect the target from attack depends on the nature of these policies. The policies must themselves be specified as properties that can be checked by viewing the target's memory pages (as is standard for rootkit detection tools [5], [39], [9], [41], [21], [40], [26]). In doing so, the policies may leverage knowledge about the target's data structures and data layout. For example, in a Linux-based target, the monitor could use data structure type definitions and addresses of root symbols as specified in the target's `System.map` file, which suffice to traverse all dynamic data structures at runtime. As another example, the target may execute its own anti-virus tools to monitor and protect its user processes from malware infection. However, malware may target the anti-virus itself, thereby undermining the security of the target. An LLM-based integrity monitor can oversee the integrity of code and data pages of the target's anti-virus tools, thereby improving overall system security. As can be seen from these examples, the target can supply a variety of integrity policies to LLM's integrity monitor, which enforces (or checks) them. These policies themselves are not the focus of our work and we elide a detailed discussion of the space of enforceable policies (although we discuss several example policies for rootkit detection in Section VI). Instead, we focus our attention on the core ability of our LLM-based mechanisms to oversee the target's attack surface.

When the monitor detects an integrity violation, we *leverage core0's ability to halt coreUs* by sending an interprocess interrupt to these cores. Control then returns to core0, which subsequently acquires control over an output device to emit suitable diagnostic information. The execution of the target can be resumed only via another interprocess interrupt from core0.

In our current design, the LLM architecture provides the monitor running on core0 the ability to inspect the contents of shared memory and the local memories of coreUs. However, the monitor *cannot* access the register state of coreUs. The inability of core0 to inspect the registers of coreUs leads to two shortcomings:

(1) *Inability to detect attacks on register state.* A rootkit can attempt to bypass the integrity monitor by directly attacking coreU registers, which are not visible to core0. As an example, consider the following *memory shadowing attack*, which maliciously modifies the `cr3` register that stores a pointer to the current page table. A rootkit can create a malicious shadow copy of the operating system at another location in shared memory, and make `cr3` point to page table of this operating system. In effect, this attack creates two copies of the operating system—the malicious shadow copy, which actually executes on coreUs, and the original benign copy, which does not execute, but simply resides in shared memory. The rootkit can also create the illusion that the original operating system is executing by periodically mirroring benign changes to its data structures. This attack can fool the monitor that executes on core0, which cannot access `cr3` and hence, has no way of determining whether the code and data that it inspects, *i.e.,* that of the original copy, actually executes on the coreU processors.
While such a hypothetical attack can bypass our system integrity monitor, it may also be possible for the monitor to employ heuristics to detect such memory shadowing attacks. For example, it could employ techniques to discover data structures in raw memory pages that it believes are unused by the operating system [14]. If the monitor identifies several "free" memory pages with data structures that are substantially similar to those of the operating system, it can conclude that a memory shadowing attack is possibly underway.

(2) *Inability to cleanly checkpoint target system.* When core0 detects that an attack is underway, it halts the execution of coreUs. However, the operating system on the coreU may be in the process of performing a critical operation (*e.g.,* writeback of a vital file system data structure) when core0 halts its execution. Because core0 does not have access to coreU registers, it cannot cleanly checkpoint the target system. As a result, the target operating system may be unable to reboot, or may lose important data when it is restarted.
It is not possible to trust any data manipulated by a rootkit-infected operating system, and it is highly desirable to reinstall a clean operating system post infection. However, there are still cases where users may want to retrieve information from a corrupted system. Although LLM does not offer support for checkpointing, it may be possible to periodically checkpoint and backup persistent storage devices so that users can still recover *some* of their data after an infection has been detected. Future work could consider alternative techniques to checkpoint the target operating system so that users can still recover their (possibly corrupted) data from the checkpoint. This checkpoint could also include additional information, such as register state of the target, so that users can infer the state of the target when it

was halted, thereby providing forensic evidence of the potential cause of infection.

V. Implementation

We created a prototype that implements the ideas discussed so far to set up a tamper-proof monitoring environment. Because LLM hardware is not currently available, we emulated its core features using the QEMU system emulator (qemu-kvm-0.12.5). Our prototype executes a Linux-2.6.26-based target on this hardware in conjunction with a monitor based on the xv6 operating system (revision 4) [13]. In this section, we focus on the details of our platform, and the changes that we made to xv6 and Linux. In the next section, we present a rootkit detector that leverages this tamper-proof environment.

Our emulated hardware platform is a 32-bit x86 machine with four processing cores: one core0 and three symmetric coreUs. The system is configured with 1 GB of physical address space. Core0 is configured to have 548 KB of local memory. We use the first 800 MB of the physical address space for our Linux-based target and its processes and the remaining 200 MB for the monitor operating system and its processes. Physical addresses in the range `0x32000000`–`0x32089000` accessed by core0 resolve its local memory, while the other addresses resolve to shared main memory.

To build the monitor operating system, we chose xv6, an instructional operating system from MIT. Our choice of xv6 was motivated by its minimal functionality and small code size (see Fig. 4), features that are essential to ensure that code in the TCB is easily verifiable. We made a number of changes and additions to xv6 to make it suitable as a monitor operating system for our LLM-based platform.

(1) We modified xv6's page fault handling mechanism. The original version of xv6 is a minimal operating system, which does not handle page faults from user space processes (*i.e.,* it expects all pages to fit in main memory and paging to disk is not supported). We changed this mechanism so that a page fault generated by the process bring in the faulting page from shared main memory into core-local memory.

(2) We added code to implement the secure paging mechanism. When xv6 receives a page fault, the secure pager calculates a SHA-1 hash value of the page, and checks it for a match against a whitelist stored in local memory before bringing it in the local memory.

(3) We added code to load the target operating system. This code is invoked at the end of the startup process, once the rest of the monitor has been initialized.

(4) We modified its memory allocation code to: (1) allocate xv6 page tables and kernel stacks within the physical address range corresponding to core-local memory to make them inaccessible to the target operating system; these data structures remain resident in core-local memory and are never "swapped" out into shared main memory; (2) allocate memory for its user processes in shared main memory.

We configured the size of core-local memory (548 KB) for our hardware platform by studying the memory footprint of the xv6 operating system and its processes. The code of an unmodified xv6 kernel itself occupies approximately 54 KB, while xv6 enhanced with our secure paging mechanism occupies approximately 84 KB. Due to its relatively small memory footprint, this code resides completely within core-local memory. Aside from 84 KB for the kernel itself, we budgeted the space in core-local memory as follows:

- 256 KB for xv6's page tables and kernel stacks;
- 128 KB to buffer pages swapped-in from shared memory;
- 80 KB to store a whitelist of hashes of pages swapped in from shared memory. Each hash is a 20 byte SHA-1 digest of a memory page. The size of this hash table can be modified, but we chose this size to accommodate hashes for code and data pages of the monitor that are stored in shared main memory. On our prototype, the monitor's code and data occupy 16 MB on shared main memory (*i.e.,* 4000 physical memory pages), so 80 KB suffices to store their hashes.

Entity	SLOC
Unmodified xv6 (revision 4)	8688
Changes to xv6	404
Secure pager	592
SHA-1 (from RFC 3174)	539
OS loader (loads target)	146
Total additions/modifications to xv6	1681

Fig. 4. Lines of code added to or modified in xv6 (revision 4) to create the monitor operating system.

In our prototype, all of the above pages are allocated and remain resident within core-local memory, thereby remaining hidden from other cores. We calculate the hashes of the monitor's main memory pages at the end of initialization and store them in core0's local memory. As discussed above, the monitor's user processes are managed in shared main memory. We use a RAM file system as the root file system to manage these user processes, which are stored as binary executables. When we load this binary for execution, the request is translated into a memory access. This access initially causes a page fault (because the page is located in main memory, rather than in the local memory), thereby triggering the secure paging mechanism to bring the corresponding pages into local memory.

We also had to make minor changes to configure Linux to boot as our target operating system. First, we configured it to allocate its data structures in the first 800 MB of shared main memory. By doing so, it avoids using the shared pages that contain monitor code and data during normal operation (though a rootkit's attempts to use these pages will be detected by the secure pager). Second, Linux typically assumes that it is the sole operating system running on the hardware platform. This assumption is violated in our platform because xv6 loads Linux, and is already executing on the system when Linux boots. Consequently, we had to modify Linux to only boot on three cores (the coreUs) of our platform, and reset the local APIC (advanced programmable interrupt controller) for the boot core, *i.e.,* the first coreU processor that loads and initializes Linux.

VI. Case Study: Rootkit Detection

Rootkits accomplish their malicious goals by modifying the code and control data of a victim operating system. Recent work [6] has also demonstrated stealthy forms of rootkits that

Rootkit	Attack description	Integrity policy
adore	Hide malicious user-space files and running processes by modifying entries in system call table to point to malicious code.	Continuously check that the system call table remains unchanged after target has initialized.
knark	Similar to adore	Similar to adore
adore-ng	Achieves same goal as adore, but does so by hooking a lookup function pointer (`proc_iops.lookup`) contained in the inode operation table of the `/proc` directory.	Check that kernel function pointers remain unchanged after initialization.
hideme	Hide malicious processes by modifying `pidhash` and `pidmap`, which are data structures of the `/proc` file system that store the PIDs of active processes. Diagnostic tools rely on these data structures to enlist processes on the system.	Traverse the list of active processes (obtained via `init_task` on Linux) and check that the PID of each entry has a corresponding entry in `pidhash` and `pidmap`.
pmap-hide [21]	Similar to hideme	Similar to hideme
enyelkm	Modify dispatch code kernel that is invoked in response to a system call. The dispatcher is modified to invoke attacker-defined code instead. Achieves same functionality as `adore` and `adore-ng` without modifying the system call table or function pointers.	Check the SHA-1 digest of Linux's text section.

Fig. 5. Summary of rootkits detected using our approach. This figure also shows the integrity policies used to detect the rootkits.

achieve their malicious goals by only modifying noncontrol kernel data, without ever executing malicious kernel-mode code. In response to these threats, a wide variety of rootkit detection techniques have been developed (*e.g.,* [39], [56], [41], [9], [40], [5], [21], [26], [20], [9], [34], [47]). We focus on rootkit detection techniques that operate by fetching the target's pages and checking that they satisfy a prespecified integrity policy (*e.g.,* [40], [21], [41], [26], [9]).

In the LLM architecture, the integrity policy checker is implemented as a user-space process that executes within the monitor. The integrity policy can be a desirable property that must always be enforced (*e.g.,* state-based control flow integrity [41], or SBCFI, which requires function pointer targets to be approved kernel code) or can be a domain-specific property supplied by a security analyst (*e.g.,* a data structure invariant that must be satisfied by kernel linked lists [40]). Depending on the integrity policy, the monitor may require various levels of understanding of the semantics of the target. For example, consider an integrity policy that requires the target code and static data pages to remain unmodified. Such a policy can be enforced with minimal semantic understanding of the target: it simply suffices to provide the integrity policy checker with the list of pages that must be checked along with a list of hashes over the contents of these pages. More complex policies may require intricate knowledge of the target operating system's data structures and data layout. SBCFI [41] and Gibraltar [5], for instance, require the monitor to traverse the target's kernel data structures. To do so, the monitor process must have a list of entry points into the target (*e.g.,* the target's `System.map` file) and data structure type declarations of the target. It can use this information to recursively traverse the target's dynamic data structures.

We considered the rootkits shown in Fig. 5 (unless otherwise mentioned, obtained from PacketStorm [2]) to evaluate LLM-based rootkit detection. Some of these rootkits were meant for older versions of the Linux kernel, so we ported their functionality to Linux-2.6.26, our target kernel. In each case, we were able to successfully detect the rootkit with the corresponding policy shown in Fig. 5.

We also evaluated the ability of the monitor to defend itself from a rootkit-infected target. We wrote a rootkit that attempts to compromise the monitor by corrupting its code and data stored in shared main memory using `memset` to write a specific value to a monitor code/data page. The secure pager was successfully able to detect this modification when the page was next accessed by core0. The SHA-1 hash of the page did not match the value saved in local memory, thereby triggering the secure pager to raise an alert.

We evaluated the performance impact of our LLM-based rootkit detector using the UnixBench 5.1.2 suite [51]. We conducted experiments with two configurations of this workload to measure overhead. In the first experiment, we configured UnixBench to run on three cores, and executed it within our target (a modified Linux-2.6.26 kernel), which itself ran on the three coreUs within our QEMU-based LLM emulator. The xv6-based monitor executes a user-process that ran every one second to scan code and data pages of the target. We also executed the same workload configuration on Linux-2.6.26, which ran on a four-core SMP emulated by QEMU. Except for core-local memory and a privileged core, the configuration of this SMP was identical in all aspects to our LLM platform. These results are reported in Fig. 6(a). In the second set of experiments, we repeated the same steps above, but the workload was configured to use four cores. Therefore, in the case of the LLM machine, where only three cores are available for normal operation (the coreUs), the workload caused contention for cores. These results appear in Fig. 6(b). We believe that the two configurations above are representative of the situation where the machine is lightly loaded, and there is no contention for processing cores, and the situation where the workload causes contention for processing cores, respectively. We therefore expect to see similar results even for an LLM machine equipped with a larger number of processing cores.

All experiments were conducted with QEMU executing on our host machine: a Dell Optiplex 755, with an Intel Core2 Quad Q6600 2.4 GHz CPU, with 2 GB memory running Linux 2.6.32. To minimize errors in our performance measurements, we removed unintended migrations of QEMU threads between the cores of our host machine using the `schedtool` utility.

Fig. 6 presents the results of our evaluation, averaged over five runs of the experiment (standard deviations are also shown). Each bar in the figure presents the performance of the UnixBench executing on LLM relative to the same benchmark configuration executing on a 4-core SMP machine. As our results show, the benefit of executing a rootkit detector on a

dedicated core is clear. When there is no contention for cores (Fig. 6(a)) the performance of the target is not affected in most cases, and induces a 4% overhead in the worst case. This small overhead can possibly be attributed to contention on the memory bus. In fact, we saw minor speedups in a few cases, which can be attributed to anomalies introduced by measuring performance within an emulated platform running on host hardware. When there is contention for cores (Fig. 6(b)), the performance degradation observed is roughly proportional to what one would expect with one fewer core, *i.e.,* performance degrades by approximately up to one-fourths on LLM hardware.

The performance overheads of an LLM-based monitor are comparable to those of systems that are equipped with a dedicated coprocessor that fetches and inspects the target's memory (*e.g.,* [39], [5]). Unlike systems that use virtual machine technology to inspect the target's memory, LLM and coprocessor-based systems have the advantage of not inducing any additional overhead on the normal execution of the target when there is no contention for processing cores. In contrast, virtualization based systems typically impose some performance overheads on the execution of the target, mainly because of the need to emulate traps in the target operating system within the hypervisor. Techniques such as paravirtualization (*e.g.,* as used in Xen) and hardware support for virtualization can potentially be used to ameliorate such performance overheads and make them comparable to those observed in LLM.

VII. Related Work

A. Isolating System Integrity Monitors

There is much prior work on developing isolation architectures for system integrity monitors. The main requirement of such architectures is the ability to set up a tamper-proof environment within which to execute the monitor. Researchers have explored the use of both virtualization and hardware support to enable such an environment.

Chen and Noble [10] were among the first to describe the advantages of using virtualization for security. Subsequently, Garfinkel and Rosenblum [20] developed virtual machine introspection (VMI), a technique to isolate security monitors from the target being monitored. In VMI, the target runs within a virtual machine (VM), while the monitor executes within another VM and observes the target VM. Garfinkel and Rosenblum also demonstrated the use of VMI to detect a rootkit-infected target. Since being proposed, numerous works have leveraged VMI for security, and several APIs [55], [53] have been implemented to ease the task of writing VMI-based security monitors. Monitors can use VMI to provide a number of security services, including enforcing code integrity (*e.g.,* [34], [47]), control-flow integrity [41], [49] and domain-specific integrity properties of dynamically-allocated data structures (*e.g.,* [40], [21], [41]).

In contrast to these works, which execute the monitor and the target in different virtual machines, and hence different virtual address spaces, the LLM architecture executes the monitor and the target on the same physical machine, albeit on different processing cores. In VMI, the monitor and the target are isolated via address space protection enforced by the hypervisor, while in LLM, the monitor is protected via a combination of the features of LLM hardware and the secure paging mechanism.

The main advantage of LLM over VMI is that it eliminates the need for hardware virtualization. This is important because commodity VMMs are often complex, resulting in a large TCB. For example, the Xen hypervisor (v4.1) has approximately 150 K lines of code, and dom0 and supporting libraries, which are also part of the TCB, can contain as much as 1.5 million lines of code [37]. This complexity can introduce numerous bugs into the TCB, as is evidenced by recent reports of vulnerabilities in the hypervisor and dom0 [15], [16], [18], [17], [3], [27], [44], which can in turn be exploited to hide malware from the monitor [31].

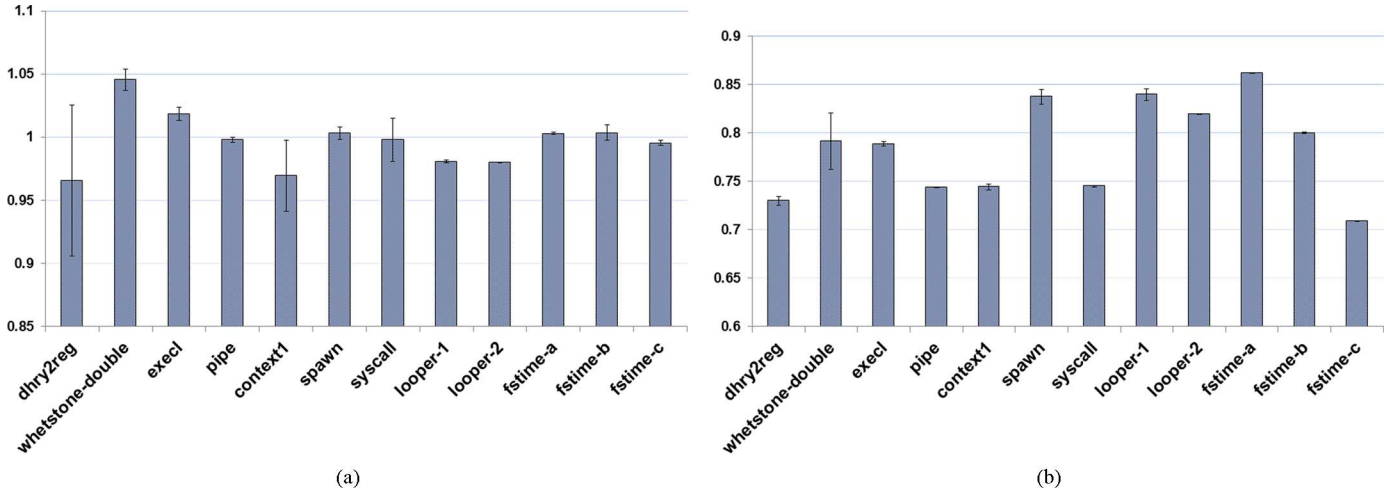
B. Reducing the TCB: Software-Centric Solutions

The large size of the TCB in modern VMMs, coupled with the discovery of vulnerabilities in them, has led to research on securing hypervisors from attack, and on techniques to reduce the size of the TCB of VMMs. IBM's sHype project uses trusted hardware to improve the assurance of the hypervisor [46], while HyperSafe [54] provides control-flow integrity guarantees within the hypervisor using a hardware technique called nonbypassable memory lockdown.

The Flicker [35] and TrustVisor [36] projects seek to reduce the size of the TCB by leveraging trusted hardware available on commodity processors [19], [12]. They set up an execution environment within which security-sensitive code blocks can execute without being tampered by malicious code. These techniques can significantly reduce the size of the TCB—to a few hundred lines of code in the case of Flicker, and to about 6300 lines in the case of TrustVisor. However, both these projects explicitly aim to protect the execution of small security-sensitive code blocks (*e.g.,* portions of an application that deal with sensitive data) in malicious environments. They also place restrictions on the content of this code, *e.g.,* these code blocks must execute with interrupts disabled.

While Flicker and TrustVisor aim to protect the execution of small blocks of code, there have also been efforts to improve the trustworthiness of VMMs by reducing the amount of privileged code in them. An example of such an effort is the Nova project [50], which proposed an architecture where the hypervisor is a few thousand lines of code, thereby bringing it within the realm of formal verification [32]. While Nova proposes a new hypervisor, the Xen disaggregation [37] and Xoar [11] projects work with a commodity VMM (Xen), and identify opportunities to reduce its TCB by employing privilege separation.

The main point of difference between these projects and LLM is that the above projects adopt a software-centric approach, *i.e.,* the task of isolating the security monitor is entrusted entirely to a software layer and this layer has to interact with the monitored system for handling VM-Exits, hypercalls and emulating privileged operations. In contrast, LLM relies on a hardware-centric approach, using local memory to isolate the security monitor from an untrusted target and security monitor runs independently of the monitored system. When the security monitor does not fit within local memory, it employs secure paging to verify that the monitor has not been tampered.



(a) (b)

Fig. 6. Performance evaluation with the UnixBench 5.1.2 benchmark suite. The graphs show the performance of each benchmark running within Linux on an LLM machine relative to their performance within Linux on an SMP machine (i.e., LLM performance/SMP performance). (a) Results with workload configured to run on three cores. (b) Results with workload configured to run on four cores. (Note: The labels `looper-1` and `looper-8` denote the commands "`looper ./multi.sh 1`" and "`looper ./multi.sh 8`," respectively. The label `fstime-a` denotes "`fstime -b 1K -m 2K`," `fstime-b` denotes "`fstime -b 256 -m 500`," and `fstime-c` denotes "`fstime -b 4K -m 8K`." Other labels denote the standard ways to execute the corresponding benchmarks in UnixBench.)

Software-centric approaches have the benefit of being able to run on commodity hardware. However, they also suffer from the pitfalls of software evolution, *i.e.,* the software-based TCB must be verified for correctness as it is modified over time to add new features. In contrast, a hardware-centric approach relies mainly on the correctness of hardware mechanisms (*e.g.,* the properties of core0) to ensure security. Hardware verification techniques do not have to deal with the complexities of software (*e.g.,* infinite state, pointers), thereby leading us to conjecture that it may be easier to formally verify a hardware-centric approach than a software-centric approach, although we have not attempted such verification of the LLM architecture ourselves.

C. Reducing the TCB: Hardware-Centric Solutions

Researchers have explored the use of hardware support to isolate the target from the monitor. Efforts to do so include secure coprocessors [56], [39] and the use of NICs such as the Myrinet PCI intelligent network cards [5], [1]. These techniques operate by physically isolating the monitor from the target (*e.g.,* by executing the monitor on a coprocessor or another physical machine) and using hardware support on the target machine to fetch memory pages via DMA. Because the target is not involved in the memory transfer, the monitor can still detect stealthy malware. However, Rutkowska has shown that such hardware-based RAM acquisition can be bypassed on AMD processors [45]. The attack operates by corrupting the memory map of the memory controller (the northbridge), thereby returning attacker-defined values in response to the monitor's requests for the target's memory pages.

The work most closely aligned in goals with LLM is the NoHype project [28], [29], which proposes a hardware architecture and software stack that provides many of the same the benefits of virtualization. Like LLM, NoHype also leverages multicore hardware to isolate virtual machines from each other. In NoHype, each VM executes on a dedicated processing core (possibly multiple cores) and is isolated from the VMs executing on other cores. NoHype partitions the physical main memory of the machine so that each VM can only access the partition assigned to it. It also configures I/O devices so as to give each VM dedicated access to a virtualized I/O device.

Although similar to LLM in its goals of eliminating the need for virtualization while using hardware support to provide isolation, the NoHype architecture cannot directly be used to construct system integrity monitors. Because NoHype partitions physical memory between VMs, it is not possible for one VM to inspect the memory pages of another. Although a privileged software layer exists in NoHype, it can only start and terminate other VMs and access devices, but cannot inspect their memory contents. This is acceptable for the NoHype architecture, because its main goal is to remove attacks that may result as a consequence of VM multitenancy in a virtualized environment (the NoHype paper provides a detailed survey of such threats). In contrast, an LLM-based monitor has access to all of shared memory, thereby facilitating memory introspection.

VIII. Conclusions and Future Work

This paper presents LLM, a hardware-centric approach to ensure tamper-proof execution of system integrity monitors. LLM-based monitors can enforce code and data integrity in a target operating system without themselves being infected by compromised or malicious targets. The unique features of the LLM-hardware combined with the secure paging mechanism proposed in this paper allow such integrity monitoring without support for virtualization. We demonstrated the utility of the LLM architecture by using it to build a rootkit detector that inspects a target operating system while itself remaining untampered by rootkits.

As presented in this paper, LLM allows security monitors to be isolated without support for virtualization. Nevertheless, we do not claim that the LLM architecture can serve as a substitute for virtualization. On the one hand, LLM's features make isolation a first-class, hardware-level primitive. On the other hand, hardware-centric solutions (*e.g.,* NoHype and LLM) cannot currently support other useful features of virtualization, such as VM

checkpointing and migration, which can readily be implemented in software-centric solutions (*i.e.,* VMMs), and are critical in cloud-based environments.

In future work, we plan incorporate a recovery mechanism in the monitor operating system. Our current prototype does not handle recovery in case of attacks against monitor. Upon detecting attacks, the monitor operating system halts the system and emit diagnostic information. This may potentially impact the availability of the system. Designing a recovery mechanism with very limited memory and the knowledge of semantics of target operating system will likely introduce new challenges.

A combination of hardware and software-based techniques can possibly offer the best of both worlds, and this is an approach that we also propose to explore in future work. We plan to explore how an LLM-based security monitor executing on core0 can ensure the integrity of a VMM executing on coreUs. In such an architecture, the VMM can itself implement integrity checking for its VMs (*e.g.,* by implementing rootkit detection) and the LLM-based monitor ensure the integrity of the VMM's code and data, (*e.g.,* by checking for the presence of hypervisor-level rootkits [31]) while itself remaining isolated from the VMM.

REFERENCES

[1] Myricom: Pioneering High Performance Computing [Online]. Available: www.myri.com

[2] Packet Storm [Online]. Available: http://packetstormsecurity.org/UNIX/penetration/rootkits/

[3] Xbox 360 Hypervisor Privilege Escalation Vulnerability 2007 [Online]. Available: www.h-online.com/security/news/item/Xbox-360-hack-was-the-real-deal-732391.html

[4] J. P. Anderson, Computer Security Technology Planning Study Deputy for Command and Management Systems, L. G. Hanscom Field, Bedford, MA, Tech. Rep. ESD-TR-73-51, 1972, vol. II.

[5] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Trans. Depend. Secure Comput.*, vol. 8, no. 5, pp. 670–684, Sep./Oct. 2011.

[6] A. Baliga, P. Kamat, and L. Iftode, "Lurking in the shadows: Identifying systemic threats to kernel data," in *Proc. IEEE Symp. Security and Privacy*, 2007, pp. 246–251.

[7] R. Banakar, S. Steinke, B. s. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. Tenth Int. Symp. Hardware/Software Codesign (CODES)*, 2002, pp. 73–78, ACM.

[8] M. Campoy, A. Perles Ivars, and J. V. Busquets Mataix, "Static use of locking caches in multitask preemptive real-time systems," in *Proc. IEEE/IEE Real-Time Embedded Syst. Workshop (Satellite of the IEEE Real-Time Syst. Symp.*, 2001.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. ACM Conf. Comput. and Commun. Security*, 2009, pp. 555–565.

[10] P. M. Chen and B. Noble, "When virtual is better than real," in *Proc. Workshop on Hot Topics in Operating Syst.*, 2001.

[11] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *Proc. SOSP*, Oct. 2011, pp. 189–202.

[12] LaGrande Technology Preliminary Architecture Specification, Intel Corporation, Intel Publication no. D52212, 2006.

[13] R. Cox, M. F. Kaashoek, and R. T. Morris, Xv6, A Simple Unix-Like Teaching Operating System [Online]. Available: pdos.csail.mit.edu/6.828/xv6

[14] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proc. ACM/USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 255–266.

[15] Xen Guest Root can Escape to Domain 0 Through Pygrub, CVE-2007-4993.

[16] Multiple Integer Overflows in Libext2fs in E2fsprogs, CVE-2007-5497.

[17] Directory Traversal Vulnerability in the Shared Folders Feature for Vmware, CVE-2008-0923.

[18] Buffer Overflow in the Backend of XenSource Xen Para Virtualized Frame Buffer, CVE-2008-1943.

[19] AMD64 Virtualization: Secure Virtual Machine Architecture Reference Manual Advanced Micro Devices, AMD Publication 33047 rev. 3.01, 2005.

[20] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distr. Syst. Security Symp.*, 2003.

[21] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," in *Proc. 16th Conf. Architectural Support for Programming Languages and Operating Syst.*, 2011, pp. 279–290.

[22] The SCC Platform Overview – Intel Research, 2010 [Online]. Available: techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf

[23] The SCC Programmer's Guide Revision 0.61 – Intel Research, 2010 [Online]. Available: techresearch.intel.com/spaw2/uploads/files/SCCProgrammersGuide.pdf

[24] "Single chip cloud computer: An experimental many-core processor from Intel labs," in *Proc. Intel Labs Single-Chip Cloud Comput. Symp.*, 2010 [Online]. Available: communities.intel.com/servlet/JiveServlet/downloadBody/5075-102-1-8132/SCC_Symposium_Mar162010_GML_final.pdf

[25] M. Jayakumar, Bootstrap Processor Selection Architecture in SMP System, Aug. 2000.

[26] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based 'out-of-the-box' semantic view reconstruction," in *Proc. 14th ACM Conf. Comput. and Commun. Security*, 2007, vol. 13, no. 2, 12 pages.

[27] K. Kortchinsky, "Hacking 3d (and breaking out of vmware)," in *BlackHat USA*, 2009.

[28] E. Keller, J. Szefer, J. Rexford, and R. Lee, "Nohype: Virtualized cloud infrastructure without the virtualization," in *Proc. Int. Symp. Comput. Architecture*, 2010, pp. 350–361.

[29] E. Keller, J. Szefer, J. Rexford, and R. Lee, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proc. ACM Conf. Comput. and Commun. Security*, 2011, pp. 401–412.

[30] Y. Kinebuchi, T. Nakajima, V. Ganapathy, and L. Iftode, "Core-local memory assisted protection (fast abstract)," in *Proc. 16th Pacific Rim Int. Symp. Dependable Computing*, Dec. 2010, pp. 233–234.

[31] S. King, P. Chen, Y. Wang, C. Verblowski, H. J. Wang, and J. R. Lorch, "Subvirt: Implementing malware with virtual machines," in *Proc. IEEE Symp. Security and Privacy*, 2006, pp. 314–327.

[32] 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. ACM Symp. on Operating Systems Principles (SOSP)*, Oct. 2009, pp. 207–220.

[33] N. Li, Y. Kinebuchi, and T. Nakajima, "Enhancing security of embedded Linux on a multi-core processor," in *Proc. IEEE 27th Int. Conf. Embedded and Real-time Computing Syst. and Applicat.*, Aug. 2011, pp. 117–121.

[34] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. 17th USENIX Security Symp.*, 2008, pp. 243–258.

[35] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. Eur. Conf. Comput. Syst.*, 2008, pp. 315–328.

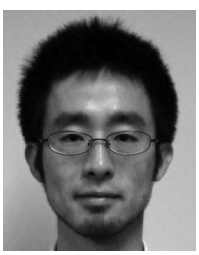
[36] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Security and Privacy*, May 2010, pp. 143–158.

[37] D. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proc. ACM Intl. Conf. on Virtual Execution Environments (VEE)*, Mar. 2008, pp. 151–160.

[38] O. Nishii, I. Nonomura, Y. Yoshida, K. Hayase, S. Shibahara, Y. Tsujimoto, M. Takada, and T. Hattori, "Design of a 90 nm 4-CPU 4320 mips SoC with individually managed frequency and 2.4 GB/s multi-master on-chip interconnect," in *Proc. IEEE Asian Solid-State Circuits Conf.*, 2007, pp. 18–21.

[39] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot: A coprocessor-based kernel runtime integrity monitor," in *Proc. USENIX Security Symp.*, 2004, pp. 179–194.

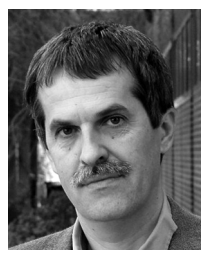
[40] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proc. USENIX Security Symp.*, 2006, pp. 289–304.
[41] N. L. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. ACM Conf. Comput. and Commun. Security*, 2007, pp. 103–115.
[42] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23rd IEEE Real-Time Syst. Symp., 2002 (RTSS 2002)*, pp. 114–123.
[43] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison," in *Proc. DATE*, 2007, pp. 1484–1489.
[44] R. Wojtczuk, "Subverting the Xen hypervisor," in *Proc. BlackHat USA*, 2008.
[45] J. Rutkowska, "Beyond the CPU: Defeating hardware based RAM acquisition, Part I: AMD case," in *Proc. Blackhat Conf.*, 2007.
[46] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. Griffin, and S. Berger, sHype: Secure Hypervisor Approach to Trusted Virtualized Systems, IBM Research, Tech. Rep. RC23511, 2005.
[47] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM Symp. Operating Syst. Principles*, 2007.
[48] 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. VEE*, 2009, pp. 121–130.
[49] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proc. Networked and Distributed Syst. Security Symp.*, 2011.
[50] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proc. ACM Eurosys*, Apr. 2010, pp. 209–222.
[51] Byte-Unixbench: A Unix Benchmark Suite, [Online]. Available: http://code.google.com/p/byte-unixbench
[52] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. 2003 ACM SIGMETRICS Int. Conf. Measurement and Modeling of Comput. Syst. (SIGMETRICS '03)*, 2003, pp. 272–282, ACM.
[53] VMsafe Partner Program, VMWare [Online]. Available: www.vmware.com/go/vmsafe
[54] Z. Wang and X. Jang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. IEEE Symp. Security and Privacy*, 2010, pp. 380–395.
[55] Xenaccess – A Virtual Machine Introspection Library for Xen [Online]. Available: code.google.com/p/xenaccess
[56] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proc. 10th ACM SIGOPS Eur. Workshop: Beyond the PC*, 2002, pp. 239–242.



Yuki Kinebuchi received the Ph.D. degree in computer science and engineering from Waseda University, where he was advised by Prof. Tatsuo Nakajima. His research interests are in operating systems, system virtualization, security, system architectures, and embedded systems. He currently works for NVidia Japan.

Shakeel Butt is a Ph.D. candidate in the Department of Computer Science at Rutgers University. He previously completed an undergraduate degree from the National University of Computer and Emerging Sciences, Lahore, Pakistan. He has done research on security issues in operating systems, cloud computing, and device drivers. Starting August 2013, he will be a research scientist at NVidia, Inc.

Vinod Ganapathy is an Associate Professor of Computer Science at Rutgers University. He received the B.Tech. degree in computer science and engineering from IIT Bombay, in 2001, and the Ph.D. degree in computer science from the University of Wisconsin-Madison, in 2007. His research interests are in computer security and privacy, software engineering, mobile systems, and virtualization.



Liviu Iftode is a Professor of Computer Science at Rutgers University. He received the Ph.D. degree in computer science from Princeton University, in 1998. His research interests are in operating systems, distributed systems, systems security, mobile and pervasive computing, and vehicular computing and networking.

Tatsuo Nakajima is a professor with the Department of Computer Science and Engineering in Waseda University. His research interests are distributed systems, embedded systems, ubiquitous computing, and interaction design. Currently, his group is working on four topics. The first topic is to develop a virtualization layer for multicore processor-based embedded systems. The second topic is to develop ambient media that are new media to help human decision making. The third topic is to develop a crowdsourcing/crowdfunding services to exploit human computation. The last topic is to develop a design framework for developing gameful services with embedding fictional stories to motivate to achieve a sustainable society.