Ensuring System Integrity using Limited Local Memory

Anonymized for double-blind review

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 hypervisor support to set up a tamper-proof execution environment. Consequently, hypervisors are part of the trusted computing base. However, modern hypervisors are complex entities, with large code bases that are difficult to verify. There have also been several recent reports of vulnerabilities in hypervisors.

In this paper, we present a new machine architecture called limited local memory (LLM), which we leverage to set up an alternative tamper-proof execution environment for system integrity monitors. This architecture leverages 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 requiring hypervisor support. 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.

1. INTRODUCTION

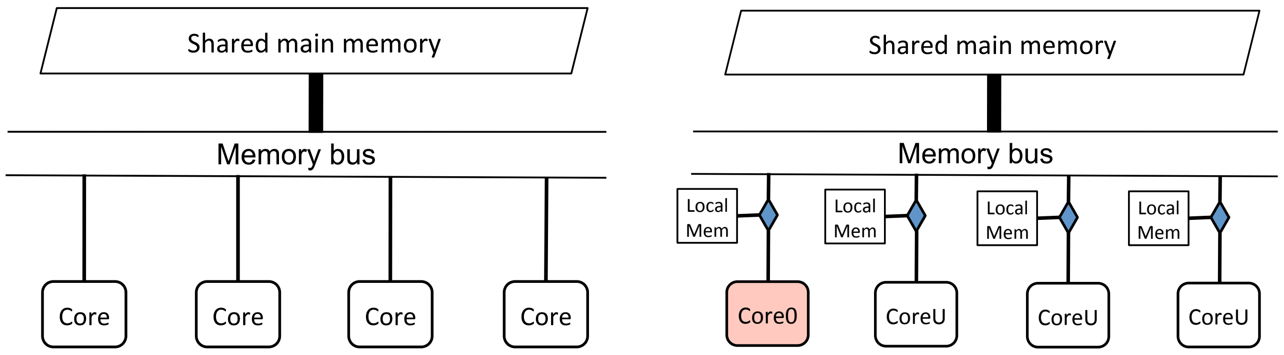
In recent years, there has been extensive research on applying virtual machine technology to problems in security. This research has been fueled both by the wide availability of virtualization, such as in the cloud infrastructure, and the attractive security guarantees provided by hypervisors. A hypervisor virtualizes system resources 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*), which allows the monitor 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.,* [10, 12, 12, 21–23, 27, 31–33, 37, 41]) 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 hypervisors to execute the rootkit detector and target operating system within different virtual machines. Hypervisors are therefore part of the trusted computing base (TCB).

However, hypervisors 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 hypervisors contain thousands of lines of code (for instance, Xen version 4.1 has approximately 150K lines of code) and exploitable vulnerabilities are routinely reported in them [5,14–17,24,34]. It is no longer reasonable to assume that hypervisors can be easily verified, as is one of the requirements for an entity that constitutes the TCB [9]. A compromised hypervisor completely subverts the security of the system, exposing the virtual machines on that platform to a variety of threats (*e.g.,* see [25,26]).

The growing complexity of hypervisors has recently motivated researchers to consider hypervisor-free hardware-based solutions that provide many of the same benefits of virtualization. An example of such an effort is the NoHype [25] project, which attempts to provide a hypervisor-free cloud-computing environment by leveraging 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 completely isolate mutually untrusted virtual machines from each other, to our knowledge it does not provide one virtual machine the ability to inspect the memory of another, a feature that is necessary for the implementation of security monitors such as rootkit detectors.

Motivated by this line of research, in this paper we investigate a hardware-centric approach to implementing security monitors such as rootkit detectors. In a manner akin to hypervisors, our approach isolates security monitors from the target system under study, which may itself be compromised or malicious. However, it does so (a) by making minor enhancements to commodity multicore hardware; and (b) by adding significantly less software to the TCB in comparison to hypervisors.

In particular, 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 recent efforts by major multicore vendors (*e.g.,* Intel [6,7] and Hitachi [30]) to equip each processing core with local storage that is addressable from that core in addition to shared main memory. We demonstrate that the LLM architecture can enable tamper-proof execution of system integrity monitors without hypervisor support. In a traditional symmetric multiprocessor (SMP) machine (*e.g.,* off-the-shelf multicore machines), all cores are *symmetric*, meaning that (1) all processing cores are equally privileged, and (2) each core can ac-



(a) Symmetric multiprocessing (SMP) architecture, as is commonly implemented on commodity multicore computers.

(b) Limited local memory (LLM) architecture, the machine model developed in this paper.

Figure 1: Comparing the SMP architecture and the LLM architecture. LLM extends SMP by adding (1) limited local memory, private to each processing core; and (2) a privileged core. Physical addresses that fall within a certain range are directed to the local memory area, while other addresses are directed to shared main memory. The privileged core (core0) can stop the execution of the other cores (coreU) and can disable each coreU from accessing its local memory area.

cess all of shared main memory in the same way. LLM modifies an SMP machine by introducing two new features:

■ **Limited local memory.** In addition to shared main memory, which is accessible from all cores, each core is equipped with a small local memory region that is accessible only by that core and not by other cores. To achieve this goal, physical memory access is modified so that when a core generates an address within a certain predefined range, that address resolves to the local memory area private to that core. The rest of main memory is shared by all cores.

■ **Privileged core.** In an LLM machine, one core is designed to be more privileged than the others. In the spirit of virtualization, we refer to this privileged processing core as *core0* and every other core as a *coreU*. This extra privilege grants core0 the ability to (1) freeze the execution of coreU processors, returning control to itself, and (2) disable local memory access to coreU processors.

We show that these two simple 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) and monitors the execution of the operating system executing on the coreUs (*i.e.,* the target operating system). The rootkit detector shares main memory with the target kernel, and is therefore able to inspect its code and data 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. Finally, LLM leverages multicore hardware, allowing the rootkit detector to operate on its own dedicated processing core (core0), in parallel with the target kernel, providing obvious performance benefits.

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 hypervisor support. Moreover, LLM can be implemented with modifications to commodity SMP architectures.

■ **Rootkit detection using LLM.** We demonstrate the utility of the LLM architecture by building a rootkit detector that leverages 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 operating system, thereby enabling low-overhead detection.

The rest of this paper is organized as follows. Section 2 introduces the basic features of the LLM architecture. Section 3 and 4 show that these features combined with secure paging can set up a tamper-proof execution environment for an integrity monitor. Section 5 describes our prototype implementation and Section 6 demonstrates its utility to detect rootkits. Section 7 discusses related work and Section 8 concludes.

2. THE LLM ARCHITECTURE

In this section, we describe and motivate the key features of our LLM architecture. We begin with a description of SMP architectures, which are popularly embodied in multicore personal computing machines today.

A traditional SMP machine consists of several symmetric processing cores that share access to physical main memory (Figure 1(a)). All physical memory locations are visible to each core, and all cores address memory in the same way. At boot time, one core first acquires control and then initializes and boots the operating system on the other cores. All cores have equal privileges in accessing memory and performing computations.

The LLM architecture modifies the traditional SMP architecture by adding two key features, as shown in Figure 1(b):

■ **Limited local memory.** Each processing core is given private access to a region of memory called *limited local memory*. This memory is part of each core's physical address space in that addresses within a certain range resolve to the local memory region rather than to shared main memory. This memory is *local* to each core because it cannot be accessed by other cores. It is *limited* in that its size is small (typically a few hundred kilobytes; see Section 5) compared to the size of main memory. Other aspects of the memory hierarchy (*e.g.,* caching) remain unchanged.

■ **Privileged core.** In an SMP machine, all cores are equally privileged. Any core can reboot the other cores; this feature is used at boot time, when one core loads the operating system and then proceeds to initialize the other cores. It is also used to recover cores from errors encountered during runtime (*e.g.,* computations executing infinite loops). On the Intel x86 architecture, rebooting is supported as INIT and STARTUP inter-processor interrupts (IPI) that force a core to restart from a specified address.

In LLM, one core (core0) is more privileged than the other cores (coreUs) in two key ways. First, *a coreU processor cannot reboot the core0 processor*. In contrast, core0 is allowed to reset or halt other processors. As a consequence, core0 must be the first processing core that is initialized at boot time, and the last core that continues to execute before shutdown. This ensures that core0 acquires control of the machine during bootup, and that it is responsible for loading the operating system on the coreU processors. This feature differentiates LLM from SMP, where cores have equal privilege, and the core that first acquires control during boot time can be changed by modifying the BIOS. Second, *core0 can disable coreU processors from accessing local memory*. If a coreU attempts to read or write local memory when access is denied, a hardware exception is raised, and is handled by core0.[1]

As we will describe in Section 3, these two features 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 acquiring control of the machine during boot time, and loads the secure paging mechanism into its limited local memory area before initializing the rest of the system. The secure paging mechanism allows core0 to securely load (on demand) the rest of the monitor from shared main memory to its local memory. It checks the authenticity of any code or data accessed by core0 before loading it into core0's local memory area. The other cores cannot access this memory and therefore cannot tamper with the execution of the integrity monitor, which can then 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 instructs core0 to halt the execution of the coreUs and raises an alert. We use core0's ability to disable coreUs from accessing local memory to to ensure that malicious software cannot evade detection from a monitor executing on core0 by hiding in coreU local memory.

Our main objective in designing the features of the LLM architecture as above was to minimally disrupt commodity multicore architectures. The two features described above (limited local memory and privileged core) can be implemented with minor modifications to an multicore processor, thereby easing the path to ultimate adoption by hardware vendors. In fact, multicore processors that implement variants of the LLM architecture have been announced by major vendors. For example, in the RP1 processor recently announced by Hitachi [30], each core of the RP1 is equipped with 152KB of local memory. The Intel's single-chip cloud computer (SCC) [6, 7], a research multicore processor, also equips each core with its own private off-chip DRAM [8, Page 52]. The main motivation behind introducing local memory in these chips is improved performance for certain parallel processing tasks, where the local memory of each core serves as a local storage area that can be accessed faster than main memory. The Hitachi RP1 differs from the LLM architecture in that even though each core is equipped with local memory, this memory is not private to the core and can be accessed remotely by other cores (making this architecture akin to a non-uniform memory access machine). In both the Hitachi RP1 and Intel's SCC chips, all cores are equally privileged, much as in a commodity multicore machine. Despite these differences, we believe that LLM can be implemented with minor modifications to the Hitachi RP1 and Intel SCC chips.

[1]For the security framework described in this paper, we only require that core0 be equipped with limited local memory. However, we introduced local memory for all processing cores to ease engineering from a hardware design perspective; hardware vendors may find it easier to design a chip in which all cores address memory in the same way. Consequently, we equipped core0 with the ability to optionally disable coreUs from accessing local memory. As Section 4 describes in detail, this feature is necessary to prevent malware from avoiding detection by hiding in coreU local memory.

3. 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).

3.1 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 compromise without itself becoming a victim of the malicious software. 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.

Several such integrity monitors have been proposed in the research literature (*e.g.,* [10, 12, 22, 23, 31–33, 41]). These include monitors that check code integrity, control data integrity, as well as non-control data integrity. While these monitors differ in the techniques that they use to detect integrity violations and in the classes of attacks that they can detect, they all rely on the ability to securely fetch code and data pages of the target for inspection. The monitor must first be able to fetch this code and data without the involvement of the target (which may itself be infected by a rootkit) before applying its policies to detect integrity violations. The core contribution of this paper is the set of *mechanisms* used by an LLM-enabled integrity monitor to fetch code and data pages and not the *policies* used to detect integrity violations. Although we present a prototype rootkit detector using LLM (Section 6), we emphasize that LLM-enabled mechanisms can be used as the basis of any integrity monitor that relies on securely fetching a target's code and data pages.

3.2 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, which loads the monitor operating system and the integrity monitor. In turn, 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 Section 3.4 and Section 5.

3.3 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, it may store pages in shared main memory. These pages 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, in addition to that of the target, *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; (2) the local memory of each core is accessible only to that core; (3) core0 can disable coreUs from accessing their local memory. This requirement is necessary to prevent rootkits from evading detection by residing in the local memory of coreUs, which is not visible to core0 (and hence to the integrity monitor).

■ *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. Note that because core0 shares main memory with coreUs, which execute the target, the monitor operating system and its user processes may themselves be infected. 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 to be clean at boot time.

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) and do not further discuss such attacks in this paper.

3.4 Monitor operation

We now discuss the operation of the monitor during different phases: (1) during startup; (2) during normal operation, when it monitors an uncompromised target operating system; and (3) 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.

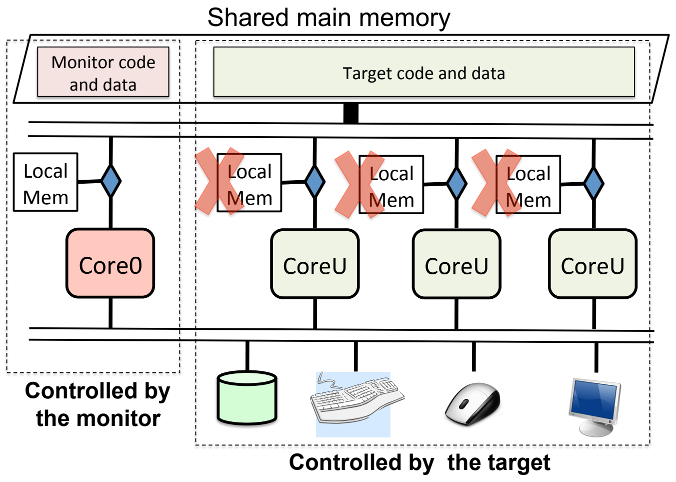


Figure 2: Configuration of an LLM machine just after startup is complete and during normal operation of the system. Core0 executes the monitor operating system and its processes from its local memory. The target executes on coreUs and controls the peripheral devices. Core0 acquires control over peripherals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

3.4.1 Startup

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 3.4.2. 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 3.4.2). The rest of the code and data of the monitor operating system are loaded into shared main memory. The monitor operating system then 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. At startup, core0 also disables local memory access to coreUs, thereby preventing the target from using local memory.

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. This 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. Note that this portion of memory is still *accessible* to the target. We just require that it should just not be used during *normal operation*. Rootkits that compromise the target may modify this memory region, in which case they will be detected by the monitor's secure paging mechanism.

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. If it 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 [19] as the monitor operating system. Figure 2 depicts the configuration of an LLM machine after startup is complete.

3.4.2 Normal execution and secure paging

Once startup is complete, the monitor oversees the execution of the target. Because the monitor and the target share 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] Local code execution. A code page to be executed by core0 must first be loaded into its local memory. Because the monitor's code may reside in shared main memory, as described earlier, it must first be copied into local memory before it can be executed on core0.

[LocalData] Local data access. A data page belonging to the monitor must first be loaded into its local memory before it can be read or updated.

[AuthLoad] Authenticate before loading. The authenticity of code and data pages belonging to the monitor must first be verified before they are loaded into 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 pre-computed 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 remains 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 bootloader revokes permissions to read, write or execute code and data pages of the monitor that are loaded into shared main memory by suitably setting permission bits in the monitor operating system's page table entries corresponding to these pages. Any attempt to access these pages results in a fault that is handled by the secure pager. In turn, the pager handles this page fault and loads the page from shared main memory to the fault address. In doing so, the pager also computes the hash of the page and checks it against the whitelist stored in local memory. If there is a match, the page is loaded into local memory and can be read, modified or executed, as the case may be. If there is no match, it triggers an alert (Section 3.4.3).

Because local memory is limited in space, pages may need to be evicted as the monitor operates. The secure pager handles such evictions by selecting the page to be evicted, computing the hash of that page and storing it in local memory, and copying the contents of the page to shared main memory. The secure pager revokes read, write and execute permissions for this page so that an attempt to again access this page will trigger a page fault. Under normal operation, this page must not be modified when it is resident in shared main memory, and the hash values must therefore match when the secure pager handles the page fault and attempts to load this page into local memory. The secure pager's authenticated load facility (computing and comparing hashes) is inspired by the techniques implemented in prior work on Patagonix [27] and SecVisor [37]. Figure 3 summarizes the secure paging mechanism.

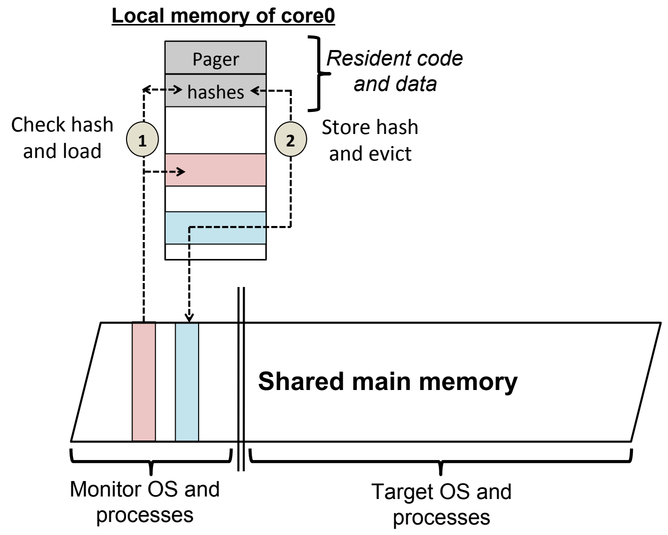


Figure 3: The secure paging mechanism. The pager is loaded into core0's local memory by the bootloader during startup. The code of the pager and the pages containing a whitelist of hashes remain resident in local memory until the machine is shut down. (1) When core0 needs to access a page that is not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

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. These are:

(**1**) pages containing 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;

(**2**) pages containing hash values of the code and data of the monitor operating system and its user processes. Some of these hash values are pre-computed (*e.g.,* those for code and static data pages) while other hash values are computed dynamically by the secure paging mechanism itself.

(**3**) pages containing 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.

3.4.3 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 fetching the target's code and data pages stored in shared memory.

The monitor raises an alert under one of three conditions:

(**1**) *the monitor's integrity is violated.* The secure paging mechanism can detect attempts by a compromised target to violate 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.

(**3**) *the target attempts to access coreU local memory.* As discussed, during startup we configure the LLM machine so that core0 disables coreUs from accessing local memory. Thus, if there are any accesses to coreU local memory at runtime, these must be because the target has been compromised. Any such access will raise an exception that will then be handled by core0.

In each of these cases, core0 issues an inter-processor 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 acquires 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). At this point, the end-user can take appropriate action, which may include restarting the machine or cleaning up the infection. We leave exploration of post-compromise user actions for future work.

3.5 Enforcing target integrity policies

As already discussed, the target can specify the integrity policies that must be applied to its code and data pages. These policies are encoded in a user process that executes on core0, and are checked by fetching the corresponding pages from shared memory.

The monitor operating system implements an `mread` system call that its user processes can use to inspect shared memory and encode a wide variety of policies. This call is akin to the interface exported by the XenAccess library [4] that allows a dom0 virtual machine to easily access and inspect domU virtual machines. For instance, the following call, when invoked within the monitor's user process, copies the contents of the target's system call table into a local buffer (`buf`) allocated in the user process' address space.

```
mread((void *) buf,
      (void *) TARGET_SYS_CALL_TABLE_ADDR,
      SYS_CALL_TABLE_LEN * sizeof(unsigned long));
```

The monitor operating system also implements a `halt_coreu` system call that instructs core0 to halt the execution of coreUs; it uses this call when it observes that the target's integrity policy has been violated.

We place no restrictions on the target's integrity policies. For example, a simple policy could check that the memory page that stores the target's system call table is not modified during normal execution [31]. More complicated policies could enforce custom invariants by traversing the target's dynamically-allocated data structures (*e.g.,* to check that function pointers point to valid code regions [33]) or to enforce invariants on non-control data structures [10, 22, 32]).

4. SECURITY ANALYSIS

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

4.1 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 therefore 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 [10, 12, 22, 23, 31–33]). 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 suffice to traverse all dynamic data structures at runtime. Because integrity policies are supplied by the target, we do not discuss them in further detail and instead focus our attention on the core ability of our LLM-based mechanisms to oversee the target's attack surface.

A monitor process running on core0 can access all pages on shared main memory and can therefore apply the target's integrity policy checks to all these pages. The only pages that are hidden from the view of the monitor are those in the local memories of coreUs. A rootkit can evade detection by locating itself in coreU local memory and can stealthily infect the operation of the target. However, as discussed in Section 3.4.1, we exclude this possibility by *leveraging core0's privilege to disable coreUs from accessing local memory*. Note that commodity operating systems executing on a multicore machine allocate all their data structures in shared main memory, which is visible to all cores. Thus, for such a target operating system, any access to coreU local memory observed at runtime is indicative of a rootkit, and the monitor can take appropriate action.

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

As discussed above, core0's privilege to disable coreUs from accessing local memory is critical to preventing rootkits that hide in local memory. Commodity SMP operating systems that are unaware of local memory can run on an LLM platform, and the performance of applications on such operating systems will remain unaffected even if coreUs are unable to access local memory. This is because these applications are also not aware of and do not use local memory.

However, processors such as the Intel SCC use local memory to enhance the performance of certain parallel workloads, and operating systems for such processors must usefully leverage local memory. Applications on such platforms may rely critically on local memory to offer good performance, and disabling coreUs from accessing local memory may be undesirable. Allowing coreUs to access local memory admits the possibility of local memory-resident rootkits.

Nevertheless, it may be possible to provide a measure of security even in this setting while only minimally sacrificing performance. Core0 could instruct coreUs to flush the contents of local memory at random intervals. While this operation may temporarily degrade the performance of applications by clearing any data or code stored in local memory, it would also flush any rootkits that reside in local

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

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

memory. We leave further exploration of this security/performance tradeoff for future work.

4.2 Ability to protect itself

The ability of an LLM-based integrity monitor to protect itself from an infected target is predicated on the three invariants discussed in Section 3.4.2. We first discuss how enforcing these invariants ensures the security 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 the target, and hence not visible to rootkits that 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 are 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 cannot be accessed or modified by the untrusted target. The security of the system is therefore bootstrapped at startup, when the bootloader initializes core0 and places pages containing the secure pager and a precomputed whitelist, containing hashes of the monitor's code and static data pages into core0's local memory, where they will reside until the machine is shut down. The extra 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, unless it detects an integrity violation, in which case it halts coreUs before acquiring control over a device to emit diagnostics.

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

5. 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) [19]. 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 1GB of physical address space, of which 548KB is local to each core while the rest is shared main memory. Physical addresses in the range `0x32000000`–`0x32089000` resolve to a core's local memory, while the other addresses resolve to shared main memory. We use the first 800MB of the physical address space for our Linux-based target and its processes and the remaining 200MB for the monitor operating system and its processes.

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 Figure 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.

■ First, we modified xv6's page fault handling mechanism. The original version of xv6 is a minimal operating system, so it is not designed to 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 instead swaps the corresponding code page from shared main memory into core-local memory.

■ Second, we added code to implement the secure paging mechanism. When xv6 receives a trap entailed by accessing a page in the shared memory, the secure pager attempts to copy the page to the core-local memory. In doing so, it calculates a SHA-1 hash value of the page, and checks it for a match against a whitelist stored in local memory, and updates the monitor's page table entry to point to the local memory page instead.

■ Third, 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.

■ Finally, 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 not "swapped" out into shared main memory; (2) allocate memory for its user processes in shared main memory (*i.e.,* address range starting from `0x32089000`).

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

■ 256KB for xv6's page tables and kernel stacks;

■ 128KB to buffer pages swapped-in from shared main memory;

■ 80KB 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 16MB on shared main memory (*i.e.,* 4000 physical memory pages), so 80KB suffices to store their hashes.

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 on this file system. 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 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 800MB 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 it to only boot on three cores (the coreUs) of our platform, and reset the local APIC for the boot core, *i.e.,* the first coreU processor that loads and initializes Linux.

6. CASE STUDY: ROOTKIT DETECTION

Rootkits accomplish their malicious goals by modifying the code and data of a victim operating system. They vary widely in the attack vectors used, and can range from simple rootkits that modify the system call table to those that hijack control flow by injecting malicious code and modifying function pointers in dynamically allocated data structures. Recent work [11] has also demonstrated stealthy forms of rootkits that achieve their malicious goals by only modifying non-control 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.,* [10, 12, 12, 21–23, 27, 31–33, 37, 41]). These techniques isolate themselves from the infected operating system, typically using hypervisors. We focus on rootkit detection techniques that operate by fetching pages from the target virtual machine and checking that they satisfy a pre-specified integrity policy (*e.g.,* [12, 22, 23, 32, 33]).

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 [33], 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 [32]). 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 [33] and Gibraltar [10], 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.

In the following examples, we illustrate LLM-based detection of several previously-known rootkits. In keeping with the primary contribution of this paper as being the LLM architecture, our main goal is to illustrate the *utility and flexibility of LLM's mechanisms* at detecting rootkits, and not the specific policies used to detect them. Our example policies below check simple invariants on specific kernel data structures. An LLM-based detector can also be used to implement more sophisticated integrity policies than the ones illustrated below (*e.g.,* SBCFI [33]).

6.1 Examples of rootkits detected

We considered the following rootkits (unless otherwise mentioned, obtained from PacketStorm [3]) for our evaluation. 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.

■ **adore** is a rootkit that hides the presence of malicious user-space files, directories and running processes. It also presents an interface to the attacker to hide/unhide a file, directory or process at runtime. It achieves this goal by modifying the entries in the system call table to point to malicious code instead.

The monitor process that detects this rootkit enforces the constraint that the system call table of the target kernel should remain unmodified for the lifetime of the system. It enforces this invariant by copying the contents of the system call table into its own address space (in an array called `systab_init`) when the target is started. It then continuously compares the saved value against the target's current system call table (located at physical address `systab_addr`). The pseudocode below shows how this is accomplished.

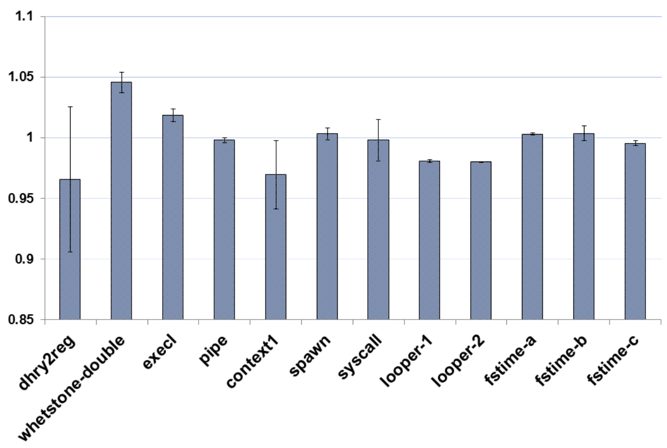
```
void *systab_init[SYSTAB_SIZE];
void *systab_curr[SYSTAB_SIZE];
mread(systab_init, systab_addr, sizeof(systab_init));
while (1) {
  mread(systab_curr, systab_addr, sizeof(systab_curr));
  for (i = 0; i < SYSTAB_SIZE; i++) {
    if (systab_init[i] != systab_curr[i])
      alert();
  }
}
```

■ **knark** also modifies entries in the system call table to achieve its malicious goals, similar to adore. We were able to detect knark using the same policy illustrated above.

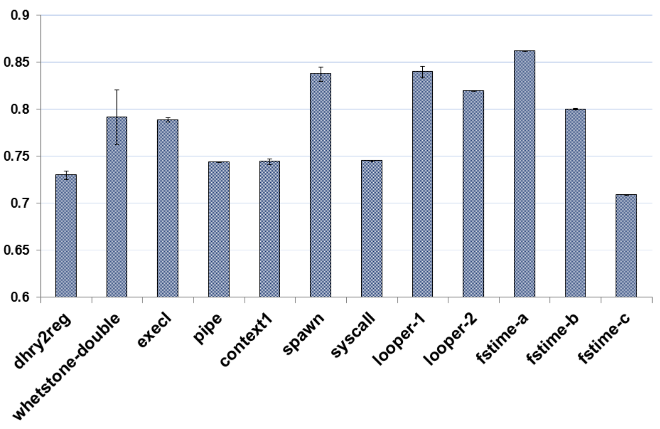
■ **adore-ng** accomplishes the same malicious goals as adore, but does so by modifying function pointers in heap data structures corresponding to Linux's virtual file system. Specifically, it hooks a lookup function pointer, which is contained in the inode operation table of the `/proc` directory object, to instead point to malicious code.

We wrote an integrity checker that detects this rootkit by ensuring that the the corresponding function pointer (`proc_iops.lookup`) remains a constant for the lifetime of the target. Although we only checked this function pointer as an illustrative example, the monitor can be extended to implement a more sophisticated policy that checks constancy of *all* kernel function pointers (akin to SBCFI).

■ **hideme** and **pmap-hide** [22] are rootkits that hides processes by altering kernel data structures used by the `/proc` file system. On Linux, each process is represented using a `task_struct` data structure, which appears as an entry of a linked list that we call `init_task`. Linux uses this linked list to schedule processes for execution. Each process also has an entry in two data structures used by the `/proc` file system: a hashtable (`pidhash`) that stores the process identifiers of active processes, and a bitmap (`pidmap`) that denotes which process identifiers are currently taken. The hideme and pmap-hide rootkits achieve their malicious goals by removing the entry of a malicious process from the `pidhash` and `pidmap` data structures, respectively. Because diagnostic tools use the `/proc`



(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.

Figure 5: 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.,* $\frac{LLM\ performance}{SMP\ performance}$).

filesystem, a running process can effectively be hidden from these tools by modifying `pidhash` and `pidmap`.

We detected these rootkits using an integrity checker that traversed the target's linked list rooted at `init_task`. We ensured that the process identifier of each entry in this list had a corresponding entry in `pidhash` and `pidmap`.

■ **enyelkm** is a rootkit that modifies the dispatch code in the kernel that is invoked in response to a system call (specifically `system_call` and `sysenter_entry` handlers). The dispatcher is modified to invoke attacker-defined code instead, thereby achieving the same functionality as `adore` and `adore-ng`, but without modifying the system call table or function pointers.

Our integrity checker uses a SHA-1 digest of Linux's text section to detect this attack, which modifies Linux's text. Our checker currently calculates a hash of the entire text section of Linux; it could instead calculate and check hashes of individual code pages as well (in a manner akin to the secure pager).

6.2 Detecting attacks against the monitor

We 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 by 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.

6.3 Performance evaluation

We evaluated the performance impact of our LLM-based rootkit detector using the UnixBench 5.1.2 suite [1]. 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 Figure 5(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 Figure 5(b).

All experiments were conducted with QEMU executing on our host machine: a Dell Optiplex 755, with an Intel Core2 Quad Q6600 2.4GHz CPU, with 2GB 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.

Figure 5 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 (Figure 5(a)) the performance of the target is not affected in most cases, and induces a 4% overhead in the worst case. 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 (Figure 5(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.

7. RELATED WORK

There is much prior work on developing 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 [13] were among the first to describe the advantages of using virtualization for security. Subsequently, Garfinkel and Rosenblum [21] 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 either within the hypervisor or 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 [4,39] 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.,* [27,37]), control-flow integrity [33,38] and domain-specific integrity properties of dynamically-allocated data structures (*e.g.,* [22,32,33]).

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 a hypervisor. This is important because modern hypervisors have increased in complexity, resulting in a large TCB. For example, the latest release of Xen has approximately 150K lines of code, and dom0 and supporting libraries, which are also part of the TCB, can contain as much as 1500K lines of code [29]. This complexity can introduce numerous bugs into the TCB, as is evidenced by recent reports of hypervisor vulnerabilities [5, 14–17, 24, 34], which can in turn be exploited to hide malware from the monitor [26].

The large size of the TCB in modern hypervisors coupled with the discovery of hypervisor vulnerabilities has led to research on reducing the size of the TCB as well as on securing hypervisors using hardware support. The Flicker project [28] seeks to reduce the size of the TCB by leveraging late launch and attestation facilities available in Intel and AMD processors [18,20]. IBM's sHype project uses trusted hardware to improve the assurance of the hypervisor [36], while HyperSafe [40] provides control-flow integrity guarantees within the hypervisor using a hardware technique called non-bypassable memory lockdown.

Aside from virtualization, researchers have also explored the use of hardware support to isolate the target from the monitor. Efforts to do so include secure co-processors [31,41] and the use of NICs such as the Myrinet PCI intelligent network cards [2, 10]. These techniques operate by physically isolating the monitor from the target (*e.g.,* by executing the monitor on a co-processor 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 [35]. 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 [25], which proposes a hardware architecture and software stack that provides the benefits of virtualization without hypervisor support. 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 hypervisor while using hardware support to retain its benefits, 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 VM (equivalent of dom0) 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 multi-tenancy 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.

8. CONCLUSIONS

This paper developed a LLM, a hardware architecture for the construction 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 a secure paging mechanism developed in this paper allow such integrity monitoring without hypervisor support. By removing the need for hypervisors, LLM eliminates the associated code complexity within the TCB. 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.

9. REFERENCES

[1] byte-unixbench: A Unix benchmark suite. http://code.google.com/p/byte-unixbench.

[2] Myricom: Pioneering high performance computing. www.myri.com.

[3] Packet storm. http://packetstormsecurity.org/UNIX/penetration/rootkits/.

[4] xenaccess - A virtual machine introspection library for Xen. code.google.com/p/xenaccess.

[5] Xbox 360 hypervisor privilege escalation vulnerability, 2007. www.h-online.com/security/news/item/Xbox-360-hack-was-the-real-deal-732391.html.

[6] The SCC platform overview - Intel research, 2010. techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf.

[7] The SCC programmer's guide revision 0.61 - Intel research, 2010. techresearch.intel.com/spaw2/uploads/files/SCCProgrammersGuide.pdf.

[8] Single chip clould computer: An experimental many-core processor from Intel labs. In *Intel Labs Single-chip Cloud Computer Symposium*, 2010. communities.intel.com/servlet/JiveServlet/downloadBody/5075-102-1-8132/SCC_Sympossium_Mar162010_GML_final.pdf.

[9] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, L. G. Hanscom Field, Bedford, MA, 1972.

[10] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(4), 2011.

[11] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, 2007.

[12] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conf. on Computer and Communications Security*, 2009.

[13] P. M. Chen and B. Noble. When virtual is better than real. In *Workshop on Hot Topics in Operating Systems*, 2001.

[14] Mitre Corp. CVE-2007-4993: Xen guest root can escape to domain 0 through pygrub.

[15] Mitre Corp. CVE-2007-5497: Multiple integer overflows in libext2fs in e2fsprogs.

[16] Mitre Corp. CVE-2008-0923: Directory traversal vulnerability in the shared folders feature for vmware.

[17] Mitre Corp. CVE-2008-1943: Buffer overflow in the backend of Xensource Xen para virtualized frame buffer.

[18] Intel Corporation. LaGrande technology preliminary architecture specification, 2006. Intel Publication no. D52212.

[19] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system. pdos.csail.mit.edu/6.828/xv6.

[20] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual, 2005. AMD Publication no. 33047 rev. 3.01.

[21] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distr. Systems Security Symp.*, 2003.

[22] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *16th Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[23] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based 'out-of-the-box' semantic view reconstruction. In *14th ACM Conf. on Computer and Communications Security*, 2007.

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

[25] E. Keller, J. Szefer, J. Rexford, and R. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *International Symposium on Computer Architecture*, 2010.

[26] S. King, P. Chen, Y. Wang, C. Verblowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symp. on Security and Privacy*, 2006.

[27] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *17th USENIX Security Symposium*, 2008.

[28] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems*, 2008.

[29] D. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Intl. Conf. on Virtual Execution Environments*, 2008.

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

[31] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symp.*, 2004.

[32] 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 *USENIX Security Symp.*, 2006.

[33] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conf. on Computer and Communications Security*, 2007.

[34] R. Wojtczuk. Subverting the Xen hypervisor. In *BlackHat USA*, 2008.

[35] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition, part I: AMD case. In *Blackhat Conf.*, 2007.

[36] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, 2005.

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *21st ACM Symposium on Operating Systems Principles*, 2007.

[38] A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *Networked and Distributed systems security symposium*, 2011.

[39] VMWare. VMsafe partner program. www.vmware.com/go/vmsafe.

[40] Z. Wang and X. Jang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[41] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *10th ACM SIGOPS European workshop: beyond the PC*, 2002.