HIMA: A Hypervisor-Based Integrity Measurement Agent

Ahmed M. Azab, Peng Ning, Emre C. Sezer
North Carolina State University
{amazab, pning, ecsezer}@ncsu.edu

Xiaolan Zhang
IBM T.J. Watson Research Center
cxzhang@us.ibm.com

***Abstract*—Integrity measurement is a key issue in building trust in distributed systems. A good solution to integrity measurement has to provide both *strong isolation* between the measurement agent and the measurement target and *Time of Check to Time of Use (TOCTTOU) consistency* (i.e., the consistency between measured version and executed version throughout the lifetime of the target). Unfortunately, none of the previous approaches provide (or can be easily modified to provide) both capabilities. This paper presents HIMA, a hypervisor-based agent that measures the integrity of Virtual Machines (VMs) running on top of the hypervisor, which provides both capabilities identified above. HIMA performs two complementary tasks: (1) *active monitoring of critical guest events* and (2) *guest memory protection*. The former guarantees that the integrity measures are refreshed whenever the guest VM memory layout changes (e.g., upon creation of processes), while the latter ensures that integrity measurement of user programs cannot be bypassed without HIMA's knowledge. This paper also reports the experimental evaluation of a HIMA prototype using both micro-benchmark and application benchmark; the experimental results indicate that HIMA is a practical solution for real world applications.**

I. Introduction

Remote attestation is a process in which a computer system proves its integrity to a remote party. It plays an important role in improving trustworthiness in distributed computing environments. Essential to remote attestation is the integrity measurement of the attested system; the integrity information allows the remote party to verify the configuration and the current state of the attesting platform (e.g., hardware and software stack), thereby to determine its trustworthiness.

There were several attempts to address the integrity measurement problem. Existing approaches (e.g., IMA [1], PRIMA [2]) rely on monitoring and measuring components that execute inside the kernel of the same system to be measured. Unfortunately, such approaches, though providing a certain degree of security, are vulnerable to runtime attacks that target the kernel. These attacks may exploit common operating system vulnerabilities to either manipulate the kernel's integrity measurement code or critical measurement data. In particular, attacks targeting measurement data (e.g., the SHA1 hash stored in memory before being sent to the trusted platform module (TPM)) cannot be easily detected by known kernel code protection methods (e.g., [3], [4]). A fundamental limitation of these approaches is the lack of strong isolation between the measurement agent and the measurement target.

Virtualization provides the required strong isolation. Terra [5] uses the hypervisor to measure the integrity of Virtual Machines (VMs) at their boot time. However, no runtime protection is provided for these VMs, as such it is subject to TOCTTOU attacks where the running guest code can differ from the initial measured code. vTPM [6] virtualizes the TPM so that IMA [1] can be ported to VMs to provide integrity measurement. Unfortunately, this approach inherits the lack of strong isolation from the measurement target.

Researchers have been looking at out-of-the-box *passive* monitoring of guests using a technique called *guest introspection*. VMwatcher [7], Antfarm [8] and XenAccess [9] are some representative efforts. Unfortunately, none of them can be trivially adapted for effective integrity measurement because they lack both *active* monitoring and memory protection, which are necessary to accommodate the event-driven nature of today's systems. Memory protection of measured programs or hooks was proposed in [10], [11]; however, similar to IMA, both techniques rely on components inside the guest VMs, thus prone to guest system vulnerabilities.

Patagonix [12] is closest to achieving the goals of hypervisor-based integrity measurement. It utilizes the hypervisor's control over the Memory Management Unit (MMU) to provide guest memory protection and identify running programs based on the hashes of individual memory pages. However, it lacks the necessary semantic knowledge about the guest VM user processes, which impacts the effectiveness of the integrity measurement. First, Patagonix cannot handle the on-demand loading of running programs to measure them in their entirety. Second, it is still possible that an attacker can craft malicious programs by reusing or duplicating code pages of legitimate programs, thus bypassing integrity measurement. Finally, Patagonix relies on a large database of trusted hashes of individual pages, which has a negative effect on its scalability. Unfortunately, It is complicated to address these issues.

The above review of past efforts testifies to the difficulty of using virtualization to provide solid system integrity measurement. Indeed, a good solution has to provide both of the following capabilities, which none of the previous approaches have, or can be easily modified to have.

This work is supported by the U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), the U.S. National Science Foundation (NSF) under grant 0910767, and an IBM Open Collaboration Faculty Award. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

- **Strong isolation** between the measurement agent and the target system. Although isolation can be achieved by using virtualization, the measurement technique must not rely on any components or hooks that can be subverted by exploiting a guest vulnerability.
- **Time of Check to Time of Use (TOCTTOU) consistency,** which guarantees the continuity of the integrity evidence beyond the measurement time. This problem arises from the fact that an attacker can change a program after it is measured and before it executes.

In this paper, we present HIMA, a hypervisor-based agent that measures the integrity of guest VMs running on top of the hypervisor, which provides both capabilities identified above. HIMA is located in the hypervisor, and performs a comprehensive "out-of-the-box" measurement of guest VMs, including both their kernels and user programs. HIMA guarantees the TOCTTOU consistency for (static) integrity measurements of guest user programs as well as the integrity of the measurements, even if the guest kernel is compromised.

HIMA uses two complementary techniques that together guarantee TOCTTOU consistency for integrity measurement: (1) *active monitoring of critical guest events* and (2) *guest memory protection*. The former monitors all events that change guest program's layout (e.g., creation of processes), so that the measurement can be performed accordingly and ensured to be up-to-date. The latter guarantees capturing any attempt to modify measured programs or to bypass the measurement.

HIMA uses the information it obtained from the guest VMs (through active monitoring) to facilitate the integrity measurement. However, the TOCTTOU consistency does not depend on this information. Instead, it is guaranteed by enforcing the policy that only measured memory pages are executable.

During the development of HIMA, we had to resolve the following challenges:

- **Challenges in active monitoring.** Although intercepting guest events can be straightforward, recognizing their impact on the state of the guest VM is non-trivial. HIMA has to intercept the system call before returning to the calling process, introspect all the needed information and overcome any guest behavior that complicates the monitoring process (e.g., reentrant kernels).
- **Challenges in guest memory protection.** Attackers can either change program binaries after they are measured or corrupt the kernel's data structures to mislead HIMA. To overcome these threats, we guarantee that measured portions of the memory will not change and unmeasured programs will not be executable. HIMA manipulates the hypervisor's control of MMU to solve these problems. Though the principle is straightforward, HIMA has its unique memory protection scheme that benefits from the active monitoring facility while guaranteeing the required semantic awareness about the state of the guest VM.
- **Challenges in semantic-aware measurement.** Semantic awareness is necessary to bridge the gap between the intercepted low-level events and the current state of the system. To produce meaningful measurements, HIMA correctly identifies the context of each event, keep an updated view of the guest state, and overcome any guest behavior that may hinder the measurement process (e.g., the on-demand loading of new program pages).

We implemented a prototype of HIMA using Xen [13], and performed a substantial set of experiments to evaluate its performance using both micro- and application benchmarks. Our micro-benchmark results show a reasonable overhead on monitored guest events (e.g., forking processes), which are infrequent and constitute a small portion of the life-cycle of common applications. Our application benchmark results further show a light impact on application execution.

Our primary contribution in this work is a collection of non-trivial engineering efforts that use the hypervisor to provide strong isolation between the measurement agent and the measurement target, and ensure TOCTTOU consistency for the integrity measurement of user programs running inside guest VMs. Among these efforts, we develop novel techniques to handle the complication involved in active monitoring, such as processing the interception of privileged events with multiple return points and reentrant kernels. We also develop techniques to overcome many of the semantic challenges that face the integrity measurement process, such as on-demand loading. Moreover, we adapt existing guest memory protection mechanisms to ensure that what's executing is indeed what's measured. Our solution suite provides a strong and semantic aware binding of measured programs and the executable permission of their code pages. As a result, HIMA guarantees TOCTTOU consistency for integrity measurement of guest VM user programs, even if the guest kernels are compromised.

The rest of this paper is organized as follows. Section II discusses the assumptions and threat model. Section III gives an overview of HIMA. Section IV explains the details of active monitoring in HIMA. Section V presents the guest memory protection techniques. Section VI gives the security analysis of HIMA. Section VII presents our performance evaluation. Section VIII discusses related work, and Section IX concludes this paper and points out some future research directions.

II. Assumptions and Thread Model

Assumptions. We assume a typical virtualized platform consisting of the hardware, a hypervisor, and a management VM. Our measurement targets are the guest VMs running on this platform. We assume the virtualized platform can attest to its own integrity using existing static measurement techniques (e.g., [1]). The same assumption has been adopted in many virtualization-based security architectures [5]–[7], [10]. Despite that the guest kernels are measured at load time, we do not trust guest kernels due to potential runtime vulnerabilities. Indeed, even if the kernel code is intact, kernels may misbehave due to compromised kernel data. We also assume that the hardware is equipped with the Non-eXecutable (NX) bit support. Guest VMs should support separating user program's code and data pages to function properly.

Threat model. We consider all attacks that require code injection or modification of running programs. We assume

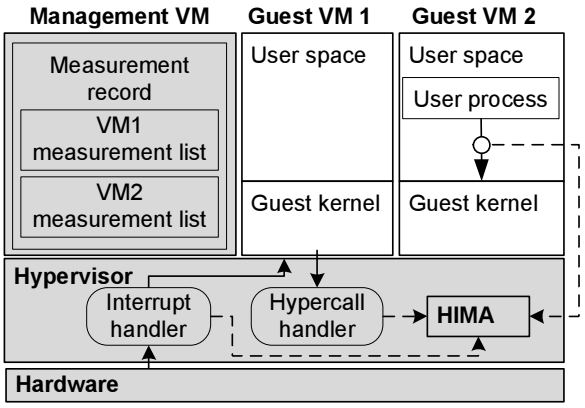

Fig. 1. HIMA architecture (Initially trusted components are in gray.)

that attackers have full access to guest VMs. In particular, attackers may gain control of the guest kernels and attempt to mislead the underlying integrity measurement. The attackers may attempt to compromise the system and then hide their traces. This raises challenges to our system, which should be valid throughout the lifetime of the guest VMs.

In this paper, we focus on the measurement of the static portion of the guest VMs. We do not consider the measurement of dynamically generated code, such as executables generated by a Java JIT compiler on an executable heap or self-modifying code. The integrity of dynamic data is also a critical issue for system integrity, which requires orthogonal techniques. We consider these out of the scope of this paper.

We target the specific objective of achieving TOCTTOU consistency for guest *user programs*. On the other hand, the protection of kernel code has been thoroughly studied in complimentary techniques (e.g., NICKLE [4]). We can safely assume that any of these techniques can be integrated with HIMA to protect guest kernel code. Finally, direct memory modification through DMA is also out of the scope of this paper as it can be simply prevented by using the IOMMU.

III. Overview of HIMA

Located inside the hypervisor, HIMA is properly isolated from the measurement targets. It generates and maintains the measurements (hashes) of the code segments of all kernel components and user programs running inside the guest VMs.

Figure 1 shows the architecture of HIMA. HIMA places hooks inside the hypervisor's code that handles guest VMs' privileged events. Thus, HIMA intercepts all these events including: hypervisor service requests (e.g., hypercalls, `VMExit`), system calls and hardware interrupts. The generated measurement lists can be stored in the management VM for the convenience of attestation or validation.

HIMA is transparent to guest VMs. It solely relies on the escalated privileges available to generic hypervisors, and can be implemented (or ported) to any hypervisor platform.

We implemented a prototype of HIMA on Xen in X86/64 architecture. Our current implementation is customized to measure para-virtualized Linux guests. Nevertheless, it can be easily modified to support other guest systems (e.g., fully-virtualized guests). In the rest of the paper, we will use our prototype as an example to clarify the details of our approach.

HIMA utilizes the hypervisor's control on guest VMs initiation to measure their booted kernels and initial loadable modules. This process is straightforward, and has been studied previously (e.g., [5]). Thus, we do not discuss it in detail. Beyond guest VMs booting, HIMA performs two major tasks to measure the guest VMs' integrity and ensure TOCTTOU consistency of measured user programs: (1) active monitoring of guest events, and (2) runtime memory protection.

IV. Active Monitoring of Guest VMs

Active monitoring allows HIMA to keep a fresh view of the memory layout of guest VMs. When a guest VM creates, terminates, or modifies its user processes or kernel modules, HIMA intercepts the relevant events and refreshes the integrity measurements. HIMA completes all its measurements before the control jumps to the loaded program to guarantee that no instruction runs inside the system before being measured.

There are several attempts to provide VM active monitoring, including Lares [10], Xenprobes [14], and VMScope [15]. However, both Lares [10] and Xenprobe [14] rely on hooks placed in the guest VMs, which can be bypassed by intelligent attackers, while VMScope [15] has significant performance overhead due to the dependency on emulation-based VMM.

HIMA performs its active monitoring differently; it is solely based on the intercepted guest events, including *system calls*, *interrupts*, and *exceptions*. HIMA faces a number of technical challenges that make active monitoring much more complicated than it appears. In the following, we describe these challenges and the key techniques used to overcome them.

A. Introspecting Event Context and Impact

In our virtualization architecture, system calls normally trap into the hypervisor before they are forwarded to the guest kernel. HIMA intercepts all system calls issued by the guest VMs, either issued using `int 0x80` or `syscall` instructions. To achieve a complete active monitoring, HIMA introspects the context of the intercepted calls and their input parameters. Moreover, it traps the calls before they return to inspect their output and assess their impact on the system state.

To introspect a guest VM event context and inputs, HIMA inspects the guest VM's registers, stack, and heap as soon as the event is trapped. Introspected information includes: (1) The event type (e.g., the system call number stored in the `EAX` register), (2) The current user process identified by the address of the base page table of its address space (the `CR3` register), (3) Event parameters stored in registers, stack or heap and (4) Both instruction and stack pointers of the running process.

As soon as the guest kernel finishes handling the event, HIMA forces it to trap into the hypervisor. For interrupts and exceptions, HIMA replaces the event's return address stored in kernel's memory with an illegal address. As soon as the event returns, this replacement causes a protection fault that traps into the hypervisor. Thus, HIMA can perform the necessary introspection and then return to the original return address.

However, we cannot use the same technique for system calls because the guest kernel can potentially rewrite a system call's return address while handling the call. One example is the `execve` system call, which loads a new program. If this call succeeds, the kernel modifies the system call's return address so that the execution returns to the new loaded program. On the other hand, if the system call fails, the kernel simply returns the error code to the calling process. It is easy to see that a trapping technique that relies on rewriting the return address will fail when such system call succeeds.

1) Handling Events with Multiple Return Points: HIMA uses debug registers [16] to address the above problem. HIMA places the return address of a system call in one of the debug registers. Consequently, the CPU inspects every instruction fetch till this specific address is fetched from the memory. At this point, a special interrupt is raised and the execution traps into the hypervisor.

To overcome the scenario where the guest kernel rewrites the return addresses of the system call, HIMA utilizes the presence of multiple debug registers per physical platform and the ability of these registers to trap write and read operations involving monitored addresses. HIMA determines the location of the system call's return address in the kernel stack. It then copies the address of this location to another debug register and set it to cause a debug exception as soon as this address is rewritten. This only happens when the `execve` call succeeds. HIMA then waits for either the write or the execute debug exception to occur in order to introspect the impact and results of the call and clear the debug registers.

2) Handling Reentrant Kernels: Our use of the debug registers to track the execution of system calls gets complicated when guest kernels support *kernel reentrancy*. Although a system call is a blocking event for user processes, they are not for the kernel. Modern kernels can relay the execution of a system call to a hardware peripheral and schedule another process to run in the meantime [17]. If the new process issues another system call, the call is handled by another kernel thread. In order to support kernel reentrancy, HIMA has to solve the following problems:

- There are a limited number of debug registers per CPU (e.g., 4 debug registers on the x86 platform) [16], which may not be enough to monitor concurrent kernel threads.
- Debug registers use virtual addresses. When a context switch occurs, the contents of the debug registers can be interpreted incorrectly in the new address space.

To overcome these problems, HIMA sets the debug registers only when a monitored execution path is currently running. Whenever a context switch occurs, HIMA either resets the debug registers or sets it with the new values required to monitor the new execution path. This approach provides HIMA with the needed flexibility of monitoring multiple events. At the same time, it minimizes the performance overhead resulting from the use of the debug registers.

To monitor guest context switching, HIMA keeps track of the kernel stack pointer. Multiple execution threads require a separate stack for each thread [17]. Hypervisors control the address of the current kernel stack to correctly forward hardware exceptions and interrupts to the active kernel thread. Upon intercepting this event, HIMA adjusts the debug registers accordingly. In our experimental prototype, kernel context switching is done via the hypercall `do_stack_switch`. Intercepting the hypercall does not violate HIMA's security guarantees, since it is the only way to do this privileged operation. To modify HIMA to measure a fully virtualized Xen guest, it will need to intercept the proper `VMExit` event.

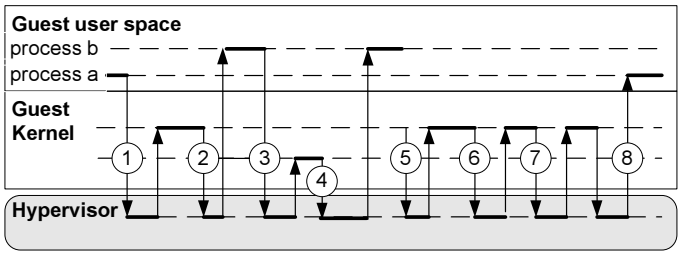

Fig. 2. Example of intercepting concurrent system calls (Thick lines denote threads running on CPU.)

HIMA keeps a list of monitored threads. Each list entry includes the values of the debug registers, the base address of the thread's stack and the event type. Whenever a context switch is captured, HIMA compares the stack address of the new thread with the stored addresses of monitored threads. If an entry is found, HIMA sets the debug registers and monitors the event until it either returns or the context changes again.

Figure 2 shows an example of handling multiple concurrent guest events. In step 1, a system call is issued by process `a`. HIMA saves the needed data and sets the debug registers. In step 2, the kernel halts the current thread to schedule process `b`. This can happen for several reasons (e.g., process `a`'s system call requires an input from a hardware peripheral). Before context switching, the execution traps into the hypervisor to correctly achieve the privileged operations accompanying the context switching. HIMA resets the debug register accordingly. In step 3, process `b` issues a system call. HIMA creates a new call entry and sets the debug registers with the new values. In step 4, the kernel finishes process `b`'s system call. A debug exception traps into the hypervisor to finish its introspection before returning to the user process. In step 5, the kernel resumes process `a`'s call upon receiving the required input (e.g., an interrupt from the hardware peripheral). (The event that caused the resumption is not shown in the figure.) Again, HIMA gets informed about the context switching and regains the saved debug register values. In step 6, the call finishes and HIMA decides to measure a memory area as a result of this call. At this point, HIMA decides to measure an executable memory based on the outcome of the system call. This measurement process (steps 7 and 8) will be explained in detail in Section IV-B.

B. Semantic-Aware Measurement

HIMA measures the program to be loaded into a guest VM after intercepting the appropriate event. To measure a program, HIMA computes the SHA1 hash of its code and initial data

segment as they get loaded into the memory. HIMA handles several situations: (1) loading kernel modules, (2) loading user programs, and (3) cloning (or forking) processes from existing ones. It is straightforward to measure kernel modules, since each kernel module is copied into the guest kernel entirely as a block of memory. Cloning or forking new processes from existing ones relies on programs already mapped into the guest VMs' memory; we present its treatment later in Section V. In the following, we focus on the second case, the semantic-aware measurement of loaded user programs.

When measuring a program, HIMA collects semantically rich information to produce meaningful measures that can be potentially used in further attestation. HIMA identifies the program's memory location and its ID (e.g., file name), and calculates the sequential hash of the whole program to identify its exact version and hence verify its integrity.

Identifying program's memory map. HIMA measures programs based on their memory mapped binaries. In order to identify a program's memory map, HIMA intercepts both `mmap` and `execve` system calls. After an `execve` call succeeds, HIMA reads the guest memory to introspect the needed information. HIMA begins introspection based on the address of the current guest kernel stack. The bottom of the stack points to the current process descriptor. The descriptor includes start and end addresses of both the code and the data segments of the new process. It also includes the address range of any pre-loaded executable libraries (e.g., the dynamic linker). On the other hand, the return of an *mmap* system call denotes the address of the mapped memory area while the area's size is one of the system call's input parameters.

Handling on-demand loading. After mapping new binaries, kernels follow an on-demand loading scheme that reads the required binaries page by page whenever a page needs to be read or executed. This behavior contradicts our intentions of calculating the hash of the entire application sequentially.

To overcome this problem, HIMA forces guest kernels to change their behavior and load the entire program before it runs. Although this may incur some performance overhead, it ensures that HIMA records the hashes of the loaded binaries before their executions to avoid any impact that may obfuscate the measurement process. To force guest kernels to load all the binaries before hand, HIMA blocks the execution of the user process and sends a series of page faults to the guest kernel. Each fault triggers a read of one page in the memory area to be measured. The return address of each exception points to a non-executable memory area to force the kernel to trap back into the hypervisor after loading the page. HIMA calculates the SHA1 hash after the entire memory area is loaded.

This process is shown in steps 6–8 in Figure 2. As soon as HIMA captures the system call return in step 6, it generates a page fault to load the first page. In step 7, after loading the page, a protection exception occurs as a result of the illegal return address and the execution traps into the hypervisor again. The process is repeated till all pages are loaded. Finally, in step 8, HIMA measures the memory area, restores original kernel output, and returns to the user process.

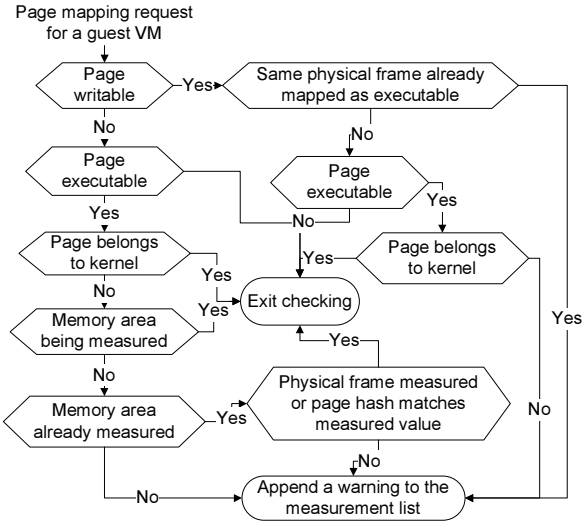

Fig. 3. Flow diagram of the proposed guest memory protection mechanism

Coupling measured programs with their IDs. After measuring a program, HIMA adds a new entry to the measurement list. Beside the hash value, the entry includes some other information (e.g., the program's file name). The program's user process and address range are also stored to help enforce guest memory protection, though the success of this enforcement does not depend on the correctness of these parameters.

V. Guest Memory Protection

Active monitoring is not sufficient to ensure TOCTTOU consistency of user programs. A runtime vulnerability in the guest kernel or user programs can be used to either bypass the monitored events and skip the measurement process or modify the measured programs before they execute. Thus, HIMA has to further guarantee that both (1) the guest VMs are only able to execute measured user programs and (2) the measured user programs cannot be modified without HIMA's knowledge.

HIMA achieves the above guarantees through guest memory protection. Specifically, HIMA utilizes the hypervisor's control of the MMU to enforce its guarantees based on memory pages' access permissions. Thus HIMA enforces TOCTTOU consistency of user programs in complete isolation of the guest VMs. However, we have to resolve a number of technical issues, including (1) identifying memory mappings of monitored pages, (2) handling remapping of measured executable pages (e.g., due to `fork` or `clone` system calls), and (3) affirming that our protection extends to actual physical memory.

Figure 3 shows the overall flow of the guest protection mechanism in HIMA. In the following, we first present the basic idea of this mechanism, and then discuss how HIMA overcomes the above technical issues.

A. Protecting Guest Memory Using Page Access Permissions

To ensure that only measured guest programs can be executed, HIMA utilizes the NX-bit page protection flag [16] (available on most hardware platforms). Executing an instruction from a page protected by this flag will cause an exception that traps into the hypervisor (and thus HIMA).

HIMA traps all guest page table updates to match the address of any executable page with those of measured programs. This step validates the information HIMA introspects from the guest kernel and eliminates the need to trust its data structures. For instance, if a program's start and end addresses are manipulated in the process descriptor to hide some malicious pages, HIMA will detect the executable mappings of these pages and invalidate the measurement.

Moreover, to prevent programs from changing after being measured, HIMA marks all measured executable pages as non-writable. If an attacker tries to write to a measured page, an exception will be raised and traps into HIMA. An attacker may try to make a measured page writable; however, HIMA disallows a page to be concurrently executable and writable.

B. Identifying Mapping of Monitored Pages

HIMA needs to differentiate between legitimate mapping of monitored pages and any other malicious page mapping. Specifically, HIMA needs to link any mapped page to one of the monitored memory areas.

HIMA identifies a mapped page by both its *address space* and *virtual address*. To identify the address space and consequently the user process, HIMA uses the address of the highest level page table pointed to by the `CR3` register. Note that identifying the virtual address of a mapped page depends on the used virtualization architecture. In our prototype using para-virtualized guest VMs, the virtual address was passed as a parameter of the `update_va_mapping` hypercall, which is the guest VM's only gateway to update the machine's page tables. In other virtualization settings like fully virtualized Xen guest VMs, identifying the virtual address will rely on the implementation details of the employed memory mapping mechanism (e.g., shadow vs. nested page tables).

To verify that a mapped page, identified by its virtual address and address space, belongs to one of the monitored memory areas, HIMA adds a few fields to the entries of the list that tracks the currently monitored events (refer to Section IV). These fields identify the address space and address range of each memory area being measured. HIMA just needs to match the mapped page with one of these monitored memory areas.

C. Handling Remapping of Measured Executable Pages

During normal operations, some of the pages that belong to measured programs may be legitimately remapped. This behavior results from one of the following cases:

1) Pages removed from the physical memory due to memory shortage and getting reloaded, from the file or swap memory, due to an access demand. These pages may be remapped to new physical frames.
2) Pages that belong to a newly forked process. These pages are usually mapped to the existing physical frames of the corresponding pages in the parent process. However, they can be remapped to new physical frames if the parent process exited or the frames were swapped out.

To verify that a page remapping is legitimate, HIMA needs to identify that (1) the page actually belongs to a valid measured memory area and (2) the page content has not changed if it was moved out of the memory.

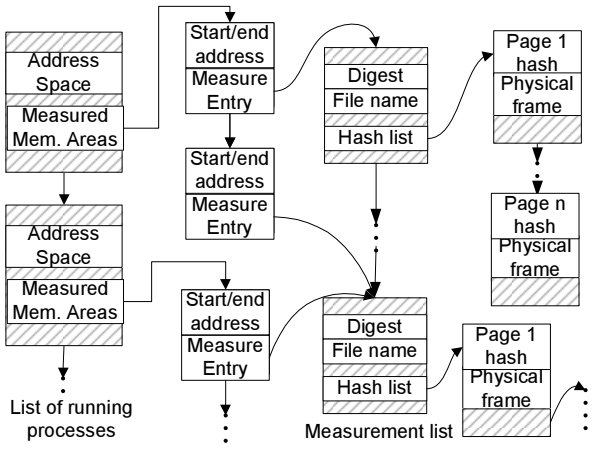

Fig. 4. Data structure for measured hash lists and guest processes

A naive approach to solving this problem would be to re-measure the hash of the whole memory area by forcing the guest kernel to load all pages as soon as an executable page is remapped. However this will cause significant performance overhead and may result in memory shortage due to reloading multiple swapped pages whenever a single page is requested.

To overcome this problem, HIMA stores the hash image of each of the pages that form a measured memory area. Whenever a page is reloaded, HIMA identifies its offset relative to the measured area and then compares its hash with the stored one. We describe this solution in detail next.

Keeping track of user processes. To validate that a mapped page belongs to a legitimate executable memory area, HIMA keeps track of the memory layout of all the processes running in the guest VM. Figure 4 shows the structure of the information HIMA keeps about running processes. Each process is uniquely identified by its address space and contains a list of all the measured memory areas in that address space.

HIMA monitors system calls responsible for process creation (e.g., `fork`, `clone`) to add new processes to the list. When a process executes a new program, HIMA updates the list of its measured areas based on the new memory layout. As shown in Figure 4, each of the entries that represent a memory area points to the corresponding entry in the measurement list which points to another list of the hashes and physical frame numbers of the pages that constitute the measured program.

Validating the integrity of remapped pages. Upon remapping an executable page, HIMA first validates that it belongs to one of the measured areas in its address space. The next step is validating the integrity of the page content. To minimize hash operations, HIMA exploits the fact that guest VMs reuse the same physical frames, which have been measured and protected, among multiple processes. As shown in Figure 4, HIMA keeps the physical frame number of each measured page. HIMA determines the offset of a remapped page in its measured memory area to retrieve the stored page hash and the physical frame number. If the page is mapped to a physical frame that was measured and never been rewritten

(explained next), HIMA can safely skip calculating the page's hash. This scenario is likely upon remapping pages to a new address space of a forked process. On the other hand, if the physical frame has been rewritten or the page is mapped to a new frame, HIMA calculates the page hash and compares it with the stored value. This scenario is likely upon reloading a swapped-out page. To determine if the guest VM has rewritten a measured frame, HIMA keeps track of all the measured frame numbers in a separate list. When any of these frames is swapped out, HIMA removes it from the list. The same list is also used for extending memory protection to actual physical frames as discussed in Section V-D.

Reducing performance overhead through caching page hashes. The same technique is used to minimize forcing guest kernels to bypass on-demand loading. HIMA keeps the hash lists of all the measured programs. If a measured program is reloaded into a new process, HIMA can validate the program's integrity based on the pre-calculated hashes of its individual pages rather than forcing the kernel to load the whole program.

Storage overhead. Despite the performance enhancement, this technique requires HIMA to store hash lists for all measured programs. However, our evaluation (Section VII) shows that the storage overhead is within acceptable ranges and can be accommodated by increasing the hypervisor's heap.

D. Extending Memory Protection to Physical Frames

As mentioned earlier, HIMA provides memory protection based on page permissions. However, these permissions apply to virtual pages rather than the actual physical frames. A physical frame can be referenced by multiple virtual mappings; this allows potential threats such as rewriting a measured page by remapping it to another writable virtual address. HIMA's operations include two extra steps to protect the physical frames of measured pages.

1) HIMA leverages the stored frame map inside the hypervisor to obtain information about the physical frames of measured pages. Our Xen prototype uses a flag that indicates if the frame has an existing writable mapping.
2) As described in Section V-C, HIMA keeps a list of all measured physical frames. The entries are also coupled with their virtual addresses. Whenever a measured frame is mapped as writable, HIMA makes sure that it does not belong to its protected pages list.

E. Writable-eXecutable (WX) Memory Regions

To avoid security exploits, kernels should allow user memory to be either executable for code or writable for data, but not both. As mentioned previously, HIMA relies on the enforcement of such W$\oplus$X mapping to provide the protection of the guest user memory.

Though most kernels support W$\oplus$X mapping, there are a few exceptions. For example, Linux kernel v2.6.18 used in our prototype maps shared writable pages as executable. Fortunately, there are orthogonal techniques that can patch such systems to enforce the W$\oplus$X mapping (e.g., PaX [18]).

On the other hand, this protection cannot be enforced on all applications. Some applications rely on WX pages to function correctly. For instance, Java converts input code into executable binaries in the heap. The lack of support for such applications is a limitation of the current version of HIMA.

VI. Security Analysis

HIMA guarantees the consistency between its measurement lists for user programs and the actual state of the guest VMs. This consistency is based on both the initial trust in the virtualized platform and the protection mechanisms that ensure TOCTTOU consistency of user programs. In this section, we first reason about the trust in guest VMs that HIMA helps achieve, and then discuss how HIMA prevents potential attacks from violating the TOCTTOU consistency of user programs.

A. Trust in Guest VMs

HIMA starts the establishment of trust in guest VMs from the initial trust in the hypervisor and the management VM. As part of our assumptions, this can be easily achieved with existing approaches (e.g., IMA [1]). HIMA builds up the trust in a guest VM by verifying the integrity of the boot-up sequence and any programs loaded in the guest VM. Specifically, HIMA measures the base kernel (possibly with the initial loadable kernel modules) and the initial user processes (e.g., the `init` process in Linux) after retrieving the binaries from the guest VM's image file. This process guarantees that once a guest VM is booted, its entire content is measured and trusted.

After booting up a guest VM, HIMA monitors all the critical events that may change its program layout, including loading and removing kernel modules (e.g., `init_module` system call), creation and termination of user processes (e.g., `execve`, `fork`, `clone`, `kill` system calls), and loading and unloading of libraries (e.g., `mmap`). It is worth mentioning here that HIMA's active monitoring can be extended to monitor any system call. In other words, HIMA can be adapted to handle any system functionalities that are not included in our prototype or any future kernel functionalities. Moreover, HIMA exploits hardware based memory protection to intercept all attempts to modify already mapped user programs and prevent unmeasured user programs from being executed. Together, these mechanisms ensure that the integrity measurement is up-to-date. In other words, HIMA guarantees the TOCTTOU consistency of the measures of all guest user programs.

B. Possible Threats and Defenses

In the following, we discuss the resilience of HIMA against potential threats that aim to inject malicious code into a monitored guest VM. By examining all the possible paths to such attacks, we find that they fall into four broad categories: (1) changing the program layout of the guest, (2) modifying the content of the measured memory, (3) executing unauthorized code, and (4) changing the guest kernel's code segment.

Security guarantee for user programs. HIMA explicitly binds the measurement and the executable permission of a page together. That is, the unmeasured code pages will not

have the executable permission. As a result, any attempt to maliciously change the program memory layout will not result in executable pages. Moreover, any attempt to modify a measured page will fail due to the non-writable permission placed on the page by HIMA. Finally, any attempt to execute unauthorized code will also fail, since the code pages will not have the right executable permissions. In other words, all attempts in the first three classes of attacks will fail due to HIMA's guest memory protection.

Note that even if a runtime data vulnerability is exploited to compromise the guest kernel, the above argument still holds, because the compromised kernel does not have control over the guest memory protection mechanism enforced by HIMA. Thus, HIMA will not allow such attacks to change any measured user programs or load new ones without its knowledge (i.e., TOCTTOU consistency). Although HIMA extracts some semantic information from the guest kernel (e.g., file name, program address range), such information is only used to facilitate the measurement process and any future attestation. As mentioned earlier, the guest memory protection does not rely on such information.

The above argument also applies to the execution of scripts. An attacker may attempt to exploit a legitimate interpreter to perform malicious actions by supplying compromised script files. HIMA can be easily extended to measure the contents of input files as they are read by the kernel. In this case, HIMA needs a specific input to identify these files and enforce the required security guarantees.

Security guarantee for guest kernels. Almost all major OS kernels need mixed code and data pages to function correctly. The behavior clearly contradicts with HIMA's memory protection mechanism, which requires W$\oplus$X mapping to protect the kernel code segment. Fortunately, as mentioned in section II, there are existing mechanisms that resolve this problem (e.g., NICKLE [4]). (Note that these mechanisms can separate the kernel code from kernel data and prevent kernel code modifications, but they cannot protect the kernel data, which can still be modified if an attack can exploit some kernel vulnerabilities.) With the presence of such a mechanism, HIMA's memory protection mechanism would right away extend to include kernel code injection or modification.

Limitations. HIMA does not handle applications that rely on writable and executable memory pages (e.g., Java, self-modifying code), as discussed earlier. There are also threats that HIMA is not intended to handle, for example, attacks that exploit data vulnerabilities either in user programs or the guest kernel. The current version of HIMA only focuses on the integrity of code running inside the guest VMs.

VII. Experimental Evaluation

We perform an extensive set of experiments to evaluate the performance of HIMA. Our experimental platform is a DELL OptiPlex 755 PC, which has a 3.0 GHz Intel dual-core processor with 4 GB RAM. Our HIMA prototype uses Xen version 3.2.1 as the base hypervisor. The management VM (i.e., Dom0 in Xen's terminology) runs 64-bit Fedora 8 Linux, and the guest VM that HIMA measures runs para-virtualized 32-bit Fedora 8 Linux. In all experiments, Xen allocates one virtual CPU and 512 MB RAM to the guest VM. Both the management VM and the guest VM use 2.6.18 Linux kernel.

A. Evaluation Methodology

Our goal is to assess the performance overhead introduced by HIMA, including both computational and storage overheads. We use both a micro-benchmark and an application benchmark. The former is used to understand the direct computational overhead introduced by HIMA on the guest kernel (the impact of intercepting the privileged operations), while the latter is to determine the overall overhead on applications.

Micro-benchmark. We use UnixBench [19] in our micro-benchmark evaluation. HIMA has three main operations that add overhead to guest kernels: (1) intercepting certain events, (2) verifying the integrity of mapped pages, and (3) measuring executable programs. The first step includes intercepting the `execve`, `mmap`, `fork`, `clone`, `vfork`, `open`, `create`, `dup` and all versions of the `exit` and `kill` system calls. HIMA also intercepts all the interrupts and exceptions generated by the hardware, particularly page faults, debug exceptions, and protection faults. The last two steps rely on calculating the SHA-1 hash, which is relatively expensive. The number of hash operations can be reduced significantly through caching the hash values of individual pages. To evaluate the effectiveness of caching, we evaluate two different versions of HIMA: with and without caching.

Application benchmark. We hypothesize that HIMA will introduce an overhead on guest applications only when they require a monitored operation. To validate this hypothesis, we choose a diverse set of five applications: two file compression applications, a web server, a download manager and a kernel compilations benchmark. The file compressing applications (`gzip` and `bzip2`) evaluate the performance incurred by extensive I/O operations, with the latter being more computationally intensive. Both programs are evaluated based on the time they use to compress an 11 MB file. The web server (`apache`) relies heavily on both threading and I/O operations. We compare the server throughput in delivering a 1.2 KB file using `ApacheBench` v2.3 with 50 concurrent requests. To eliminate networking overhead, we ran the benchmark on the same guest VM that runs the web server. To verify that HIMA does not incur any overhead on network operations, we use the download manager `prozilla` to download a 600KB file from a local server. We ran this experiment over a weekend in order to avoid any overhead added by the network. Finally, to evaluate the performance of applications that are computationally extensive and rely on threading and file operations at the same time, we use the Linux kernel compile benchmark `kcbench`.

Storage overhead. We estimate the amount of heap memory needed by HIMA to store the necessary guest state and cached hashes. We run all our benchmarks, both the micro- and application benchmarks, on a single running instance of the

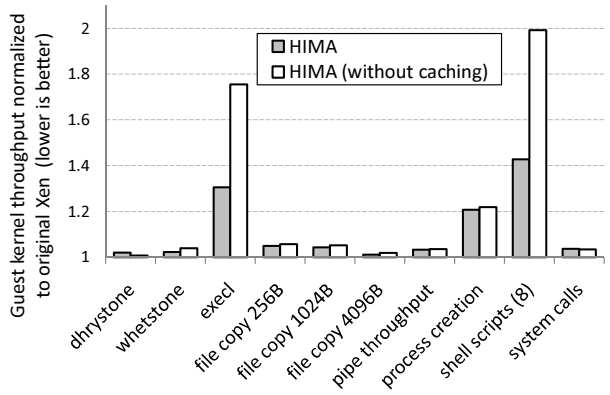

Fig. 5. Computational overhead with UnixBench

guest and calculate the maximum amount of memory required by HIMA.

B. Result of Micro-benchmark Evaluation

Figure 5 shows the result of micro-benchmark evaluation. HIMA introduces less than 5% overhead in all test cases except for `execl`, process creation, and shell script tests. These tests measure the ability of the machine to do extensive computation and I/O operations that do not include monitored events other than the one-time loading of the test programs.

The `execl` test exposes HIMA's overhead for monitoring and measuring program-loading events. The test consists of a loop that runs a dummy program. When caching is enabled, HIMA introduces a 31% overhead. It includes a one-time measurement of the whole program (and the needed libraries) and page-level hash comparisons with cached values. When caching is disabled, HIMA remeasures the test program every time, and the overhead jumps to 75%.

Another test that incurs a large overhead is process creation through forking. Forking is a monitored operation that results in newly mapped pages to be verified. When caching is enabled, HIMA introduces 21% overhead. Disabling caching does not add much overhead, because the forking process stays in memory till the child loads, and these processes usually share the same physical frames.

Finally, the shell script test represents HIMA's worst case scenario. It consists of both loading new programs (`bash` shell) and forking processes to run concurrent tests. Moreover, the shell needs more libraries than the simple `execl` test. With caching, HIMA's overhead is 43%. The overhead rises dramatically to 99% when caching is disabled due to frequent measurements of the shell program and needed libraries.

C. Result of Application Benchmark Evaluation

Figure 6 shows the application benchmark results. For `gzip` and `bzip2`, HIMA introduces 1.25% and 0.17% overhead, respectively. As expected, `bzip2` introduces less overhead due to its computational intensive nature. The `apache` server introduces 4.64% overhead. Based on the micro-benchmark results, we can deduce that this overhead is mainly due to forking multiple processes to handle concurrent requests. The overhead by `prozilla` is negligible as the test's bottleneck

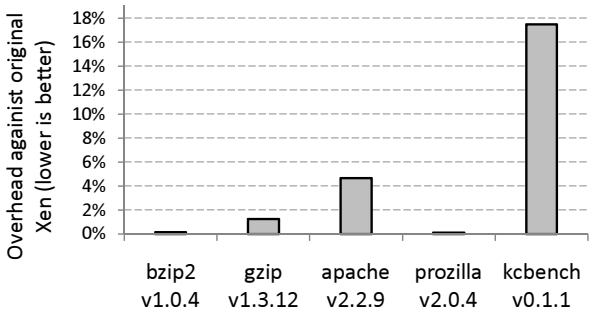

Fig. 6. Computational overhead using application benchmark

is the network operations rather than any of the monitored events. This test verifies that HIMA does not introduce much overhead on networking operations.

Although kernel compiling is not a frequently used application, we choose it to represent the worst case scenario. The operations of `kcbench`, rely on a huge number of concurrent processes. These processes run different programs to compile, link and manage kernel code files (e.g. `gcc`, `cc1`, `rm`). Consequently, HIMA monitors the creation of each of these processes and validates the integrity of all programs that run inside them. As the lifetimes of these processes are relatively short, the monitored operations represent a significant portion of their life cycle. Hence, this test shows a relatively higher overhead of 17.48%.

D. Storage Overhead

We monitored the memory required by HIMA to run all the evaluation programs. The maximum memory needed during the evaluation process was around 2.6 MB. Almost half of this memory is consumed by storing page hashes. Considering the fact that Xen's heap size is configurable and physical memory is a relatively abundant resource on today's computing platforms, we believe that the memory demand can be accommodated easily. The space used for caching page hashes could be more controversial, since this space may continuously grow and cause memory starvation if the guest VM operates for a long period of time. Fortunately, HIMA can take advantage of any cache management technique to efficiently use a pre-allocated amount of memory.

VIII. Related Work

We have discussed in the introduction previous research on integrity measurement and approaches that may be adapted for hypervisor-based integrity measurement. In this section, we discuss other related work.

Integrity measurement is the foundation of remote attestation. Multiple approaches have been developed to provide integrity evidence to a remote party by using hashes of static memory regions [1], runtime environments [20], security policies [2] or program input/output [21]. In this paper we demonstrate that HIMA enhances the original IMA measurement scheme to provide more timely and trustworthy integrity measurements by reducing the reliance on the guest OS to obtain measurements. We believe that HIMA can be extended

in similar ways to provide the same benefits to many other attestation schemes.

Several approaches have been proposed to protect the integrity of guest kernels using hypervisors [3], [4]. These techniques can be integrated with HIMA to ensure the guest kernel code integrity, thereby further improving the overall system integrity. Note that although these techniques are effective against kernel code-injection attacks, they alone are not sufficient to guarantee the runtime integrity of the entire system and hence the measurement agent due to other non-control-flow attacks against the kernel [22]. HIMA, on the other hand, is immune from all attacks that attempt to mislead integrity measurement because of its strong isolation from the guest VMs and the TOCTTOU consistency protection.

Researchers have used VM introspection in many applications including intrusion detection [7]–[9], [23]–[26], verification of access control policies [27], and general system monitoring [14], [15]. HIMA uses similar VM introspection techniques to intercept events that are related to integrity measurements. While this line of research addresses different problems from ours, our research benefits from their results.

IX. Conclusion

In this paper, we presented the development of HIMA, a hypervisor-based agent that measures the integrity of guest VMs that run on top of the hypervisor. HIMA is located in the hypervisor, and performs a comprehensive "out-of-the-box" measurement of the guest VMs, including both the guest OS kernels and their applications. HIMA provides two complementary mechanisms: active monitoring of critical guest events and guest memory protection. The former guarantees that the integrity measures are refreshed whenever the program layout in a guest VM changes (e.g., creation or termination of processes), while the latter ensures that the integrity measurement of user programs cannot be bypassed without HIMA's knowledge. With these two mechanisms, HIMA ensures TOCTTOU consistency of the integrity measurement of user programs. We have implemented a prototype of HIMA based on the Xen hypervisor. Our security analysis and extensive experimental evaluation indicate that HIMA provides a practical solution to integrity measurement.

Our future work is two-fold. First, we will refine the current HIMA implementation, enriching its support of popular guest operating systems. Second, we will investigate its application to remote attestation, preferably guest transparent attestation in virtual computing environments.

References

[1] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[2] T. Jaeger, R. Sailer, and U. Shankar, "Prima: Policy-reduced integrity measurement architecture," in *Proceedings of the 2007 ACM workshop on Scalable trusted computing (SACMAT '06)*, June 2006.

[3] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[4] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1–20.

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 193–206.

[6] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Proceedings of the 15th USENIX Security Symposium*. USENIX, August 2006, pp. 305–320.

[7] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 128–138.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006, pp. 1–1.

[9] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Annual Computer Security Applications Conference*, 2007, pp. 385–397.

[10] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, 2008, pp. 233–247.

[11] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "TOCTOU, traps, and trusted computing," in *TRUST*, 2008, pp. 14–32.

[12] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *SS'08: Proceedings of the 17th conference on Security symposium*, 2008, pp. 243–258.

[13] "Xen," http://www.xen.org/, accessed in August 2009.

[14] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for xen virtual machine," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007, pp. 1–14.

[15] X. Jiang and X. Wang, ""Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots," in *Proceedings of the 10th Recent Advances in Intrusion Detection (RAID 2007)*, 2007, pp. 198–218.

[16] I. Corporation, "Software developer's manual vol. 3: System programming guide," June 2009.

[17] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., November, 2005.

[18] PaX, http://pax.grsecurity.net/.

[19] Tux.Org, http://www.tux.org/pub/tux/benchmarks/System/unixbench/.

[20] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, 2004, pp. 13–13.

[21] E. Shi, A. Perrig, and L. van Doorn, "BIND: A fine-grained attestation service for secure distributed systems," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[22] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of USENIX Security Symposium*, August 2005.

[23] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[24] K. Kourai and S. Chiba, "Hyperspector: virtual distributed monitoring environments for secure intrusion detection," in *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005, pp. 197–207.

[25] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 91–104.

[26] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2008, pp. 91–100.

[27] M. Xu, X. Jiang, R. Sandhu, and X. Zhang, "Towards a vmm-based usage control framework for os kernel integrity protection," in *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, 2007, pp. 71–80.