Journal of Information Assurance and Security 4 (2009) 124-132

Invariant Evaluation through Introspection for Proving Security Properties

Fabrizio Baiardi[1], Dario Maggiari[1] and Daniele Sgandurra[2]

[1]Polo G. Marconi, University of Pisa
Via dei Colli, La Spezia, Italy
{baiardi, maggiari}@di.unipi.it

[2]Department of Informatics, University of Pisa
Largo B. Pontecorvo, 3 Pisa, Italy
daniele@di.unipi.it

Abstract: *Semantics-driven monitoring* discovers attacks against a process by evaluating invariants on the process state. To increase the robustness and the transparency of semantics-driven monitoring, we propose an approach that introduces two virtual machines (VMs) running on the same platform. One VM runs the monitored process, i.e. the process to be protected, while the other one evaluates invariants on the process state each time a process invokes a system call. The evaluation of invariant exploits an *Introspection Library* that enables the monitoring VM to access the memory and the processor registers of the monitored VM.

After describing the overall architecture of the proposed approach, we focus on the Introspection Library and the problems posed by the introspection of variables in the memory of a program running in a distinct VM to evaluate invariants. A first prototype implementation is also presented together with a set of performance results.

Keywords**:** intrusion detection, virtual machine introspection, invariants, process self, anomalous behavior**.**

1. Introduction

Virtualization is becoming increasingly popular, because enterprises can reduce the total number of servers by migrating the real environments, hosted on physical machines, to virtual machines (VMs) running on a single server. Since a VM is an exact replica of a real machine, it can execute the same operating system (OS) and applications but several VMs can share a single physical host concurrently. The consolidation of multiple server environments onto a single platform harnesses unused computing power and cuts hardware and lifetime costs.

While these advantages cannot be neglect, we believe that virtualization should also be exploited to build more robust systems [4, 7, 16, 28, 29]. As an example, the Virtual Machine Monitor (VMM), i.e. the software that runs, confines, and manages the VMs, exports a control interface that enables a monitoring VM to analyze the current state of other VMs through Virtual Machine Introspection (VMI) [6]. The introspection capability leverages the fact that a VM completely encapsulates the state of the corresponding physical host, so that a monitoring VM can analyze in full detail a host running inside another VM. As an example, the monitoring VM can search for specific values in the memory of another VM or inspect the content of the VCPU registers. In this way, a monitoring VM can analyze and compare the data obtained through VMI against those returned by invocations to the OS of a monitored VM. Any discrepancy signals that an attacker may have altered the OS of a monitored VM [1]. Furthermore, by analyzing the status of the VCPU of a monitored VM, the monitoring VM can access register values to retrieve the parameters of a process system call and check their correctness.

This paper proposes a semantics-driven approach to discover attacks against a process *P* based upon the evaluation of invariants on the state of *P* each time *P* invokes a system call. This approach requires two tools: a static tool that analyses *P* program to pair each system call with an invariant, and a run-time tool that intercepts the system calls that *P* issues and evaluates the corresponding invariants. Each invariant is paired with a system call invocation and it constrains the values of the process variables and of the parameters of the system call. The model underlying the definition of our framework is focused on those attacks that modify the behavior of a process as expressed in the source code, such as those that inject and execute malicious code.
In this paper we focus on the run-time tool, which is implemented by two VMs: a *Monitored VM* (Mon-VM), which runs *P*, and a monitoring *Introspection VM* (I-VM). The I-VM runs an *Assertion Checker* that evaluates invariants on the state of *P*. The Assertion Checker accesses the values of *P* variables and the processor registers of the Mon-VM through an *Introspection Library*, which has a low-level access to each component allocated to the Mon-VM, such as the main memory. Every time *P* issues a system call, the Mon-VM transfers control to the I-VM, which:

(i) retrieves the value of the processor registers of the Mon-VM that store the system call number and parameters;
(ii) determines the invariant paired with the system call that *P* has issued;
(iii) retrieves the values of *P* variables that the invariant refers to;
(iv) evaluates the invariant and kills *P* if the invariant is false, because this signals a successful attack against *P*.

Our framework does not require any modification to the source code of *P*, which is statically analyzed to compute the

Received June 10, 2009

invariants paired with each system call invocation. Instead, the system call handler of the Mon-VM kernel has been hijacked to transfer control to the I-VM every time *P* issues a system call. In this way, the monitoring is both fully transparent to *P* and highly robust. In fact, any process *P* is not aware of being monitored and the monitoring is implemented by a VM that is separated from the one that executes *P*. The current version of the run-time tool is not fully transparent because a module in the Mon-VM kernel traps the system calls and notifies them to the I-VM. The integrity of this module is periodically checked by the I-VM through VMI. Furthermore, our solution does not introduce any additional hardware units.

The rest of the paper is organized as follows. Section 2 briefly outlines some features of Xen memory management, which we have exploited to implement the current prototype. Section 3 describes the overall architecture of the run-time tool, the Introspection Library and the Assertion Checker. Section 3 evaluates the overhead of the current prototype. Section 4 discusses related works. Finally, Sect. 5 draws some conclusions and outline future developments.

2. Xen Memory Management

This section recalls some features of Xen memory management that influences the overall architecture of the run-time tool.

Besides robustness, a main goal of our framework is *fully transparency*, so that the monitoring of an application should not require any updates to the application itself or to the underlying OS. Transparency also increases robustness because the monitoring cannot be avoided even by attacking the underlying OS. The basic technologies we adopted to achieve transparency and robustness are *virtualization* and *introspection*. Virtualization introduces the virtual machine monitor (VMM), which is a thin software layer that runs on top of a physical machine and that creates, manages and monitors virtual machines (VMs). Each VM is an execution environment that emulates, at software, the behavior of the underlying physical machine. Virtual machine introspection (VMI) extends the VMM so that a privileged VM can analyze any data structure of any other VM in full detail. VMI does not require any hardware support and it offers full system visibility because the VMM can access every VM component, such as the main memory or the processor's registers. According to these considerations, the run-time tool is built around two VMs: the monitored VM (Mon-VM), which executes *P*, and the introspection VM (I-VM), which monitors *P* through VMI. The Mon-VM transfers control to the I-VM each time *P* reaches an invocation *i* of a system call. The I-VM evaluates *A(P, i)*, i.e. the invariant paired with *i* of *P*, through VMI. Assertions can involve each component of the Mon-VM, for example any program variable. A further benefit of this solution is that the tools that implement the checks run at the user-level on the I-VM and this strongly simplifies their implementation with respect to a kernel-level solution.

Xen [3] is the virtualization technology that has been chosen, mainly because of its high performance and complete integration with the Linux kernel. Xen is an open source VMM (or hypervisor) that supports the virtualization of machine hardware resources and their dynamic sharing among OSes running inside several VMs, or *Domains* in Xen terminology. Xen adopts a para-virtualized approach because the guest OSes have to be modified to run on top of the VMM. However, by exploiting the hardware support for virtualization Xen can also execute unmodified guest OSes. Xen provides isolated execution for each Domain, preventing failures or malicious activities in one Domain from impacting other ones. *Domain0* is a privileged VM because it can directly access hardware resources and configure and create other Domains to run guest OSes. In the following, we will use the term VM instead of user Domain and privileged VM instead of Domain0.

A first consequence of the para-virtualized approach is that each time a guest OS updates the memory mapping of a process, Xen has to intercept the operation to prevent a VM from interfering with another one. To deal with memory virtualization, one the most complex task for a hardware-level VMM, Xen considers three distinct issues:

(i) physical memory management, e.g. how to avoid memory fragmentation;
(ii) virtual memory management, e.g. how to minimize the overhead introduced when a VM is scheduled;
(iii) page table (PT) management, e.g. how to validate each memory access to satisfy the isolation requirement among VMs.

To give to guest OSes the illusion of a contiguous address space, Xen defines two distinct address spaces: *Machine memory*, i.e. the total amount of physical memory of the host that runs Xen, and *Pseudo-Physical memory*, i.e. the set of addresses as seen inside a VM. Two tables implement the mapping between the two address spaces: *Machine-to-Physical* (M2P), which maps the physical memory pages into pseudo-physical pages, and *Physical-to-Machine* (P2M), one for each domain, which implements the reverse mapping. The size of M2P is proportional to the physical memory, whereas the size of a P2M is proportional to the memory allocated to each VM. To minimize the performance degradation of VM context switching due to TLB misses, the topmost 64MB (for 32 bit architecture) of the virtual address space of each process contains a mapping for the Xen hypervisor itself. As far as concerns the management of PTs, there are two possible solutions: shadow PTs or direct management of the PTs by guest OSes. Shadow PTs require that a guest OS implements virtual PTs that are not visible to the MMU. In this case, to prevent interferences among VMs, Xen traps each access to the virtual PTs and propagates their updates to the real PTs used by the MMU. Direct management of PTs requires that guest OS PTs are read-only so that the OS has to invoke Xen through *hypercalls* to update the mapping.

3. Architecture of the Run-Time Tool

Our semantics-driven approach detects attacks against a process *P* by evaluating invariants, e.g. assertions, that constrain the values of *P* variables at each system call invocation. System call sites are one of the most appropriate choices for program points to check the values of the variables, because they are the points where the monitored system switches from user-level to kernel-level. Therefore, this is the most critical time when an attacker may exploit some vulnerability to gain control of the system. Invariants are the output of a static analysis of the program source code. To be fully integrated with the run-time tool, the static tool is focused on the generation of an invariant for each system call that relates values of programs variables and of system call parameters. Then, this invariant is paired with the PC value when *P* executes the system call.

As previously mentioned, the architecture of the run-time tool is based upon the cooperation of two distinct VMs: the *Monitored VM* (Mon-VM), which executes the monitored process *P*, and the *Introspection VM* (I-VM), which verifies the integrity of *P*. The I-VM runs an *Assertion Checker* process that evaluates invariants on *P* state and accesses *P* variables through VMI. Even if the Assertion Checker can monitor several processes concurrently, for the sake of simplicity, we assume that *P* is the only process that is being monitored.

The input of the Assertion Checker is a set of invariants of the form:

```
PC, {var name: addr: type}, {expr on vars}
```

where:

- `PC` is the program counter paired with a system call;
- `{var name: addr: type}` is a set of variable names, their virtual address and their type;
- `{expr on vars}` is a set of relations among variables with the following structure:
  - `<var (OP var)* REL value >`, where `OP` is an arithmetic/logic operator and `REL` is a relational operator, such as: `a > 10; a + b >= 0; i == 5;`
  - `<var (OP var)* REL var >`, such as: `a + b > c; c == d`.

The Mon-VM kernel transfers control to the I-VM every time *P* invokes a system call. Then, the I-VM freezes the execution of the Mon-VM and the Assertion Checker invokes the Introspection Library to retrieve the current PC of *P* and the values of the variables of *P* that the invariant paired with the current PC refers to. If the invariant is false, the I-VM kills *P*, otherwise it resumes control of the Mon-VM from the current PC.

By pairing an invariant with each call, we can detect non-control-data attacks [18], [19]. Invariants can either be deduced by monitoring the program execution [20] or computed by applying a static tool to *P* source code [21]. Any non-empty assertion is the conjunction of assertions in the following classes:

- *Parameters assertions*. They express data-flow relations among parameters of distinct calls, e.g. the file descriptor in a `read` call is the result of a previous `open` call.
- *File Assertions*. To prevent symlink and race condition attacks, they check, as an example, that the real file-name corresponding to the file descriptor belongs to a known directory.
- *Buffer length assertions*. They check that the length of the string passed to a vulnerable function is not larger than the local buffer to hold it.
- *Conditional statements assertions*. They prevent *impossible paths* [22] by relating a system call and the expression in the guard of a conditional statement. As an example, in `if(uid == 0) then syscall`$_1$ `else syscall`$_2$, we pair the assertion `uid == 0` with `syscall`$_1$, to prevent a normal user from executing the same call of the root user. They may also check that the current return address matches the call issued by *P*.

3.1 Introspection Library

The Introspection Library is invoked by the Assertion Checker whenever *P* issues a system call. The library implements two introspection functions, namely *Memory Introspection*, to access the memory of a monitored VM both at the user and at the kernel level, and *VCPU-Context Introspection*, to retrieve the state of the Mon-VM virtual processor.

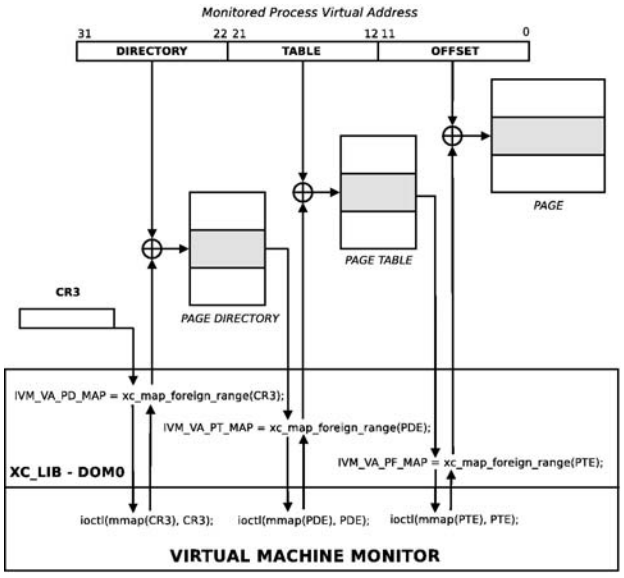

Figure 1. User-Space Address Translation

3.2.1 Memory Introspection

The implementation of memory introspection poses three

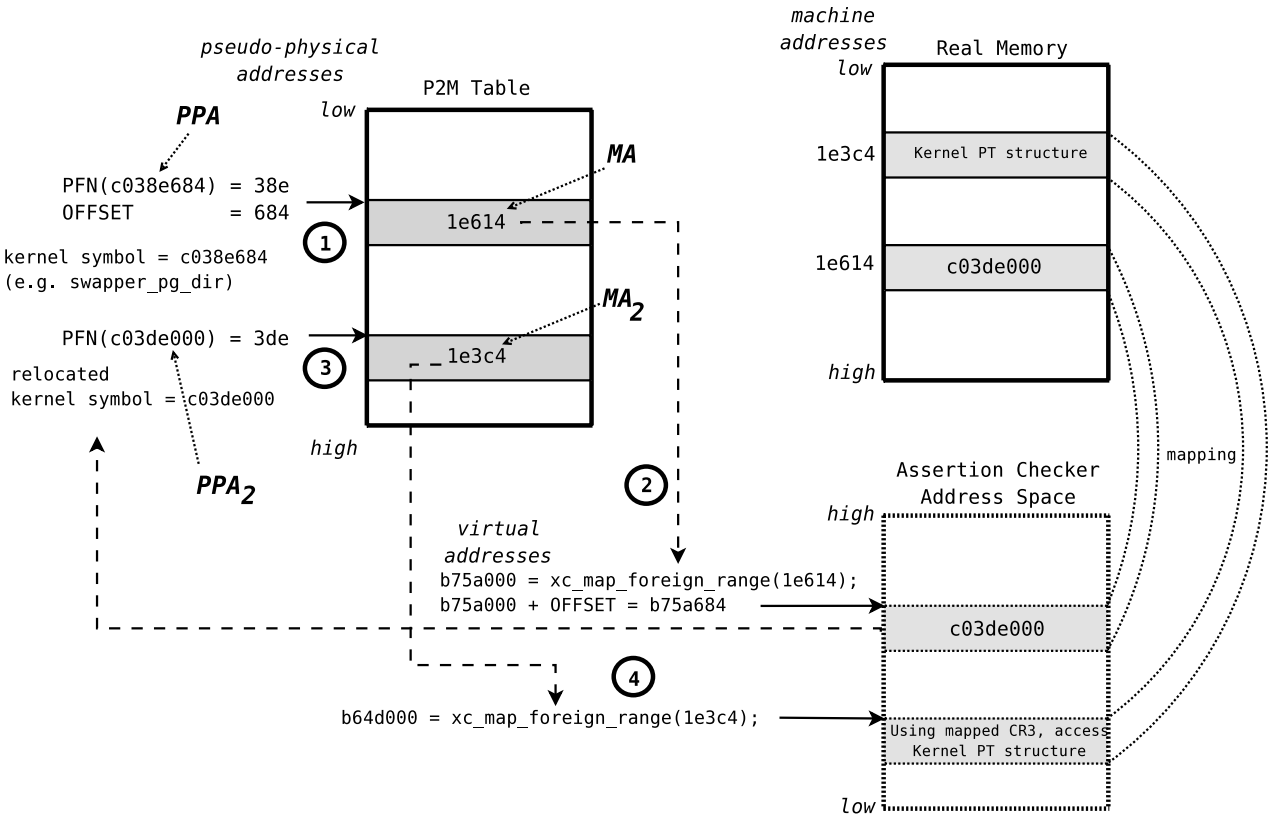

Figure 2. Kernel Static-Addresses Translation

main problems:

(i) distinguish among the various kinds of addresses introduced by Xen when virtualizing memory and correctly translate them;

(ii) directly manage accesses to the PTs of both user processes and the kernel;

(iii) map into the Assertion Checker address space the memory areas of the Mon-VM allocated either to a process or to the kernel.

To implement user-space memory introspection, the library needs to access any physical memory location allocated to the Mon-VM that corresponds to a virtual address of *P*. To translate a virtual address, the Introspection Library directly accesses the PTs of *P* and then follows the pointer to walk the paging levels to retrieve the pairing between a virtual and a physical address. In the case of para-virtualization, the addresses in all the page levels and in the registers of a virtual context of a VM are machine addresses, such as the page directory address in the `cr3` register. This implies that the Introspection Library has to map three pages to translate a virtual address into a machine address and maps the corresponding page using the `xc_map_foreign_range()` function, as shown in Fig. 1.

Conversely, Xen manages static addresses in a para-virtualized OS, such as those paired with the kernel-exported symbols, as pseudo-physical addresses. To this end, when Xen starts a VM, kernel static addresses are relocated, and the original addresses are managed as pseudo-physical ones. Hence, to translate a pseudo-physical address *PPA* paired with a kernel symbol, the Introspection Library executes the following steps (see Fig.2):

- translate *PPA* into a machine address *MA* using the P2M table. Note that *MA* does not reference the kernel symbol because it is relocated, i.e. Xen adds a further level of indirection to the kernel pseudo-physical addresses;
- request Xen to map the page at the base address of *MA* and retrieve from the resulting offset the relocated pseudo-physical address PPA_2 of the kernel symbol;
- access the P2M table to translate PPA_2 into the corresponding machine address MA_2;
- request Xen to map the page at the base address of MA_2 into the address space of the Assertion Checker process. This page stores the kernel data structure pointed to by the kernel symbol.

As soon as the Introspection Library has mapped the page that stores the pointer to the kernel page directory, referenced to by the `swapper_pg_dir` symbol, it can translate pseudo-physical addresses by accessing the kernel PTs in the same way of a process virtual address. In this case, the Introspection Library maps three pages instead of executing the previous four steps.

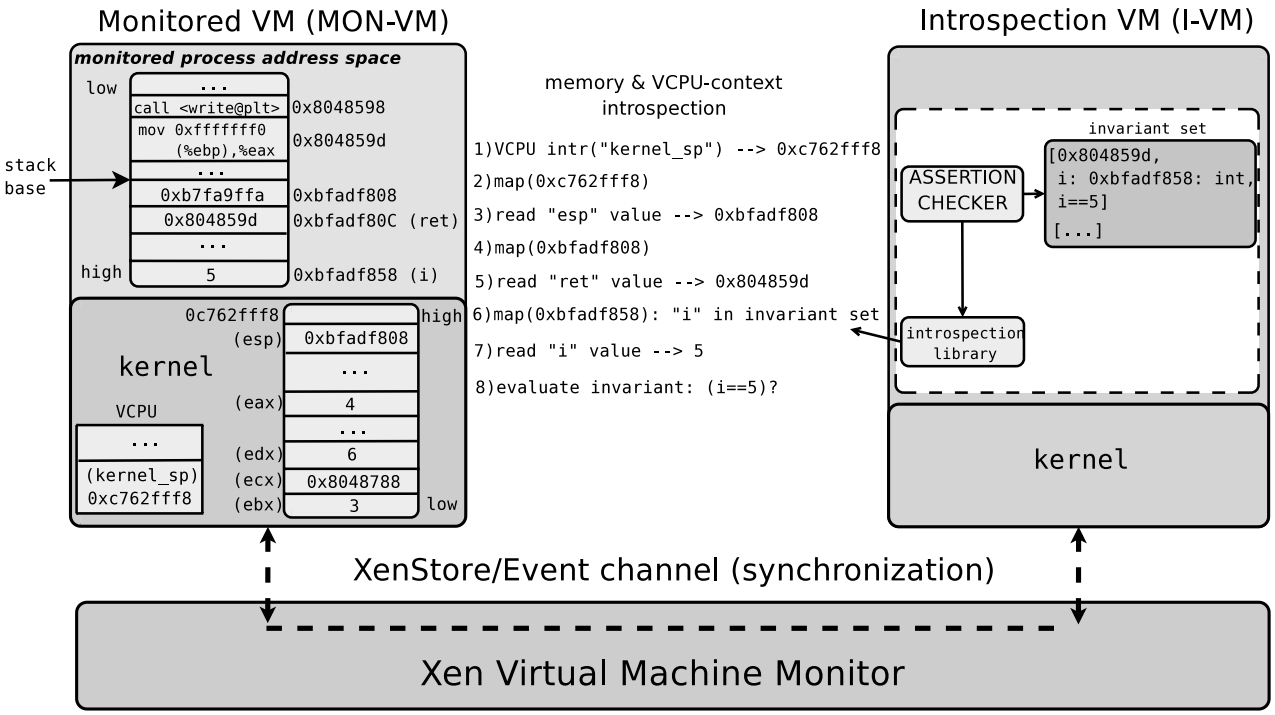

Figure 3. Invariant Evaluation

Finally, when exploiting processor virtualization extensions, Xen applies the shadow PTs mechanism and both the page directory and PTs store pseudo-physical addresses. Each time a PT needs to be updated, Xen propagates the update to the real PT, which is known to the MMU. To this end, the Introspection Library exploits the Xen `page_array` structure, which stores the pairing between pseudo-physical frame numbers and machines frame numbers.

To optimize the translation of variables allocated in the same page, the Introspection Library implements a TLB-based software solution to keep track of the pairing among virtual addresses and machine addresses for the monitored process. The optimization exploits a table that stores (`Monitored Virtual Base Address, Mapped Virtual Base Address`) pairs and where each pair records the association between a virtual base address of the monitored process and the virtual address of a machine page in the address space of the Assertion Checker. Before translating a virtual address, the Introspection Library searches this table for the virtual address of the page including the virtual address of *P*. If the address is found, then the page of *P* is already mapped in the memory of the Assertion Checker.

3.2.2 VCPU-Context Introspection

The VCPU-Context introspection enables the Assertion Checker to monitor, and modify, the content of any Mon-VM register. To support the context switch between two VMs, Xen saves the values of the CPU registers in a *Virtual CPU-Context* paired with each VM. When a VM is going to be scheduled, the current values of the registers are saved into the VCPU context of the running VM, while the registers of the new VM are restored from the proper VCPU context. The VCPU-Context is implemented by the `vcpu_guest_context_t` data structure, which contains the following fields:

- `unsigned long ctrlreg[8]`, i.e. the control registers for the virtual CPU. As an example, the control registers can be used to access the page directory through the `CR3` register;
- `struct cpu_user_regs user_regs`, i.e. the user registers, such as the EIP and all the registers used to save the parameters of a system call.

The Introspection Library has been implemented and tested on 32-bit x86 architectures both with Regular Paging and Physical Address Extension (PAE), in the two cases of para-virtualized OS guest or full-virtualized VMs.

3.3 Run-Time Invariant Evaluation

The current run-time architecture exploits two tools: (i) a kernel module in the Mon-VM to hijack system calls issued by *P* and (ii) the Assertion Checker. Each time *P* invokes a system call, the kernel module traps the call and, before servicing it, it informs the Assertion Checker. The Assertion Checker evaluates any assertion paired with the current system call. The Assertion Checker accesses the Mon-VM memory and its VCPU register through the Introspection Library. Since the interactions between the Assertion Checker and the kernel module in the Mon-VM have to be synchronous because the state of *P* cannot be updated during assertion evaluation, the Assertion Checker freezes the execution of the Mon-VM and resumes it only if the assertions are satisfied. To accomplish this, the hijacking

sys_exit	sys_mknod	sys_setfsgid	sys_setfsuid	sys_read
sys_chmod	sys_lchown	sys_setresgid	sys_write	sys_vhangup
sys_symlink	sys_mkdir	sys_open	sys_stat	sys_chown
sys_ioctl	sys_close	sys_lseek	sys_setgid	sys_ftruncate
sys_waitpid	sys_getpid	sys_setgroups	sys_flock	sys_creat
sys_mount	sys_setresuid	sys_brk	sys_link	sys_fchown
sys_rename	sys_reboot	sys_unlink	sys_setuid	sys_fchmod
sys_swapoff	sys_chdir	sys_setregid	sys_setreuid	sys_stime
sys_delete_module	sys_mlock	sys_settimeofday	sys_setdomainname	sys_truncate
sys_setrlimit	sys_ioperm	sys_sched_setparam	sys_swapon	sys_mlockall
sys_nice	sys_sethostname	sys_socketcall	sys_syslog	sys_rmdir
sys_dup2	sys_nfsservctl	sys_kill	sys_setpriority	sys_adjtimex
sys_umount	sys_sysctl	sys_sched_setscheduler	sys_quotactl	sys_exec
sys_time				

Table 1. Traced System Calls

kernel module and the Assertion Checker are synchronized through an *event channel*, which is a Xen data structure that emulates the interrupt mechanism. At startup, the Assertion Checker allocates a new event channel and waits for notifications from the kernel module: each notification corresponds to a system call that *P* wants to issue. Since just a few calls in the Linux kernel can be exploited to attack a process [23], the kernel module only hijacks the system calls listed in Tab. 1.

To evaluate the invariants, the Assertion Checker exploits the VCPU-Context introspection capability of the library to retrieve the current PC of *P* and to map the pages storing the variables of *P* into its address space to fetch their values. At runtime, when *P* issues a system call, control is hijacked and transferred to the I-VM, where the Assertion Checker:

(i) reads the current PC;
(ii) reads the current value of the variables to evaluate the invariant paired with PC;
(iii) evaluates the invariant.

During static analysis a set of invariants is paired with any address paired with a system call. Since at run-time the address found in the PC register is paired with the current system call, the Assertion Checkers needs a way to retrieve the PC at the system call site, since this is the address deduced during static analysis and paired with an invariant. The easiest way to do this is to retrieve the current system call return address (more precisely, the system call handler return address) because it points to the system call site PC+1. This address is located in the user stack. But, since the Mon-VM is in kernel space, the ESP register points to the kernel stack not to the user stack. Thus, the Assertion Checker needs to retrieve the value of the saved ESP register in the kernel stack. Then, it locates the return address in the user stack.

In more detail, the Assertion Checker (see Fig. 3):

1 accesses the VCPU context to read the `kernel_sp` register, which points to the top of the kernel stack;
2 maps the kernel stack;
3 reads the value of the `ESP` register, which points to the base of the user stack;
4 maps the user stack;
5 locates in the user stack of *P* the return address of the system call. Since the offset of the return address from the stack pointer depends upon the system call type, the Assertion Checker reads the `EAX` register to identify the system call;
6 after reading the return address, which is paired with a set of invariants, the Assertion Checker maps the pages storing the variables paired with this return address;
7 reads the value of the variables;
8 evaluates the invariant.

If the invariant is satisfied, the Assertion Checker resumes the execution of the Mon-VM, otherwise it kills *P*.

4. Performance Results

Since only the run-time tool has currently been fully implemented, we manually build the invariants to be evaluated at run-time. Furthermore, after compiling the source program, we manually search inside the object code for invocations to the `libc` system call wrapper routines or for direct invocations of system calls through the `int $0x80` mechanism. In this way, we pair each system call invocation with its return address to uniquely identify both the invocation and the corresponding invariant. In the current prototype, some modifications are required to the source code to retrieve the run-time address of any variable referred to by an invariant, its name and type. This information is stored into the *XenStore* database and can be accessed by the Assertion Checker to retrieve the value of *P* variables. In a future version, the static tool will compute variable addresses by considering both the frame pointer (EBP register) and the debugging information in the object code as well.

As far as concerns the monitoring performance, the most critical operation of an invariant evaluation is the time to access the variables. The average time to map a page of *P* into the Assertion Checker address space is about 50 μsecs. For each system call, at least two pages have to be mapped for, respectively, the kernel and the user stack. Moreover, at most one further page for each variable that the invariant refers to has to be mapped. Thus, the corresponding

overhead is at least 150 μsecs, because each invariant refers at least one variable. By exploiting the software TLB, this time can be considerably reduced anytime several program variables are stored in the same page (see Fig. 4). In this way, the Assertion Checker can access the variables without mapping further pages of the virtual address space of *P*. If the variables are stored in the same page, each access requires 20 μsecs. Therefore, since the software TLB can also be exploited to access the kernel stack, anytime the Assertion Checker has to retrieve the value of just one variable, the overhead due to the mapping and the evaluation is about 60 μsecs for each system call. Taking into account the rate of system call invocations, the overhead of the execution time is lower than 10% on the average. This overhead can be justified because the monitoring is fully transparent and because of the robustness due to the cooperation between distinct VMs.

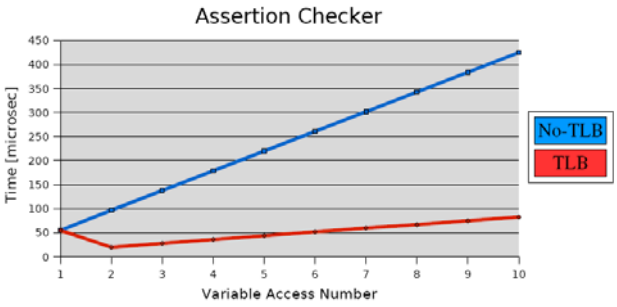

Figure 4. Time to Retrieve a Variable Value

5. Related Works

An approach to VM introspection is proposed in [13] with *XenAccess*, a monitoring library for OSes running on Xen, which provides virtual memory introspection and, as opposed to our library, virtual disk monitoring capabilities. It exploits the XenControl library to access the memory of a VM from a distinct one and applies introspection in a way that is similar to our approach. *XENKimono* [24] detects security policy violations on a kernel at run-time, by checking the kernel from a distinct VM through virtual machine introspection. XenKIMONO proposes 2 strategies: (i) integrity checking, to detect illegal changes to kernel code and jump-tables (system call table, IDT, page-fault handler); (ii) cross-view detection to detect the malicious modifications to critical kernel objects. Moreover, it monitors critical processes, detects suspicious activities and applies a white-list based detection, such as a list of applications that can have root access, a list of network ports that the applications can bind to and a list of kernel modules that can be loaded into the kernel. *Lares* [14] is a security tool that can actively control an application running in a guest VM by inserting hooks into the execution flow of the process. These hooks transfer control to another VM that checks the monitored application using introspection and security policies. The *guarded model* [15] combines control flow and data flow analysis by generating and propagating invariants to detect mimicry attacks. This model exploits the control flow graph where system calls are guarded by invariants, which are properties about system call arguments, return values, input variables and the values of branch predicates. *KernelGuard* [25] is a prevention solution that blocks dynamic data kernel rootkit attacks by monitoring kernel memory accesses using VMM policies. It preemptively detects changes to monitored kernel data states and enables fine-grained inspection of memory accesses on dynamically changing kernel data. For each kernel data structure that needs to be protected, a policy is written which describes how the VMM should identify the data structure in a raw view of memory as well as the characteristics of an attack against that data structure. In addition, the policy describes the pointers within the kernel's memory that point to the data structure so that those can be tracked and protected as well. At runtime, the VMM finds the data structure in memory and intercepts all writes to its address in order to validate them and ensure they do not violate the policy. [10] presents an architecture model to protect host-based intrusion detectors. It exploits the confinement provided by a VMM to separate the IDS from the monitored OS so that it cannot be subverted by intruders. With respect to our approach, this system provides a learning mode, such as the one in [5] to build a database that stores the sequences of invoked system calls. In the monitoring mode, an intrusion detection module compares data received from the VM against the data in the database. *Xenprobes* [12] is a framework to probe several Xen guest kernels simultaneously and that allows developers to implement their probe handlers in user-space. [8] demonstrates a technique to debug guest kernels of Xen VMs through *gdb*. A limitation of this approach is that it cannot debug arbitrary devices. This stems from Xen architecture, because Xen guest OSes can only see the hardware devices that Xen emulates. PAID [9] is a compiler-based framework that derives system call patterns from an application source code. *Null system calls* are inserted into the application source code to increase the amount of information available at run-time. *Manitou* [17] is a system implemented within a VMM that ensures that a VM can only execute authorised code by computing the hash of each page before executing the code it includes. Manitou sets the executable bit for the page only if the hash belongs to a list of authorised hashes. *Lycosid* [27] is a VMM-based hidden process detection and identification service. Lycosid uses cross-view validation to detect maliciously hidden OS processes by comparing the lengths of process lists obtained at a low (trusted) and a high (untrusted) level. If the trusted list is longer than the untrusted list Lycosid concludes that at least one process has been hidden. Lycosid introduces a new technique called *CPU inflation* that transparently placing patches in guest program code and, by forcing processes to run more frequently and more aggressively than they normally would, CPU inflation effectively increases the resolving power of Lycosid's identification techniques. The *VIX Tools* [26] are designed to allow an investigator to perform live analysis of a guest VM from a privileged VM. VIX consists of a library of common functions, and a suite of tools which mimic the behavior of common Unix command line utilities, such as `ps, lsmod, netstat, lsof, who, top`. The basic approach taken by these tools is to pause the target virtual machine, acquire the data necessary to perform the requested function using read-only operations, and then un-pause the target VM. Using this approach VIX can ensure that the state of the VM does not

change during the data acquisition process, and that the state of the VM is not modified while its execution is suspended.

6. Conclusion and Future Developments

We have proposed a semantics-driven approach to monitor program execution that exploits the virtualization technology to access the process memory and evaluate an invariant for each system call. We believe that this is a very general and selective strategy to detect attacks, because it can discover malicious updates to data structures anytime a process requires a critical operation to the OS. Moreover, the isolation of the monitoring VM from the one that runs the monitored process increases the overall robustness.

An area of future research is the automatic extraction of invariants from the application source code of the monitored process. Since we are currently focused on the run-time aspects of the monitoring, to evaluate both its efficacy and efficiency, the deduction of invariants through a static analysis has been formally specified through a data-flow framework [2]. In particular, we are interested in relationships among system call parameters. Moreover, we are currently working on a strategy that integrates static and dynamic information to compute the addresses of local variables, whose address may vary from run to run, by keeping track of the value of the frame pointer.

References

[1] Fabrizio Baiardi and Daniele Sgandurra. Towards High Assurance Networks of Virtual Machines. In Proceedings of the 3rd European Conference on Computer Network Defense (EC2ND 2007), Heraklion (Greece), Lecture Notes in Electrical Engineering, Vol. 30 Siris, pp. 21-34, 2007.

[2] Darren C. Atkinson, William G. Griswold, Implementation Techniques for Efficient Data-Flow Analysis of Large Programs, icsm, pp.52, 17th IEEE International Conference on Software Maintenance (ICSM'01), 2001.

[3] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A.Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In Proceedings of the ACM Symposium on Operating Systems Principles, pages 164-177, October 2003.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev., 36(SI):211–224, 2002.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy, pages 120–128. IEEE Computer Society Press, 1996.

[6] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proc. Network and Distributed Systems Security Symposium, February 2003.

[7] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. SIGOPS Oper. Syst. Rev., 39(5):91–104, 2005.

[8] N. A. Kamble, J. Nakajima, and A. K. Mallick. Evolution in kernel debugging using hardware virtualization with xen, In Proceedings of the 2006 Ottawa Linux Symposium (Ottawa, Canada, July 2006).

[9] L. C. Lam. Program Transformation Techniques for Hostbased Intrusion Prevention. PhD thesis, Stony Brook University, December 2005.

[10] M. Laureano, C. Maziero, and E. Jamhour. Protecting hostbased intrusion detectors through virtual machines. Comput. Netw., 51(5):1275–1283, 2007.

[11] F. Leung, G. Neiger, D. Rodgers, A. Santoni, and R. Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. Intel Technology Journal, 10(3):167–178, August 2006.

[12] A.-Q. Nguyen and K. Suzaki. Xenprobes, a lightweight user-space probing framework for xen virtual machine. In USENIX Annual Technical Conference, pages 15–28, 2007.

[13] Payne, B.D.; de Carbone, M.D.P.; Wenke Lee, Secure and Flexible Monitoring of Virtual Machines, Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, vol., no., pp.385-397, 10-14 Dec. 2007

[14] Bryan D. Payne, Martim Carbone, Monirul Sharif, Wenke Lee, Lares: An Architecture for Secure Active Monitoring Using Virtualization, sp, pp.233-247, 2008 IEEE Symposium on Security and Privacy (sp 2008), 2008

[15] H. Sa¨ıdi. Guarded models for intrusion detection. In PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security, pages 85–94, New York, NY, USA, 2007. ACM.

[16] Hirano, M. and Okuda, T. and Kawai, E. and Yamaguchi, S., Design and Implementation of a Portable ID Management Framework for a Secure Virtual Machine Monitor, Journal of Information Assurance and Security, Volume 2, pages 211--216, 2007

[17] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, pages 6–11, New York, NY, USA, 2006. ACM.

[18] S. Chen, J. Xu, E. C. Sezer, P.Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In Proc. of the 14th USENIX Security Symposium, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[19] S. Bhatkar and A. Chaturvedi. Sekar., R. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. Proc. of the IEEE Symposium on Security and Privacy, 2006.

[20] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. Science of Computer Programming, 69(1-3):35–45, 2007.

[21] G. Ammons, R. Bod´ık, and J. R. Larus. Mining specifications. SIGPLAN Not., 37(1):4–16, 2002.

[22] D. Wagner and D. Dean. Intrusion detection via static analysis. In Proc. of the 2001 IEEE Symposium on Security and Privacy, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[23] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In Proc. of the 7th ACM conference on Computer and communications security, pages 174–183, New York, NY, USA, 2000. ACM.

[24] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, pages 276-283, New York, NY, USA, 2007. ACM

[25] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring, Proceedings of 4th International Conference on Availability, Reliability and Security (ARES 2009), Fukuoka, Japan, March 2009

[26] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. SIGOPS Oper. Syst. Rev., 42(3):74-82, 2008.

[27] 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, pages 91-100, New York, NY, USA, 2008. ACM.

[28] M. Pollitt, K. Nance, B. Hay, R. C. Dodge, P. Craiger, P. Burke, C. Marberry, and B. Brubaker. Virtualization and digital forensics: A research and education agenda. J. Digit. Forensic Pract., 2(2):62-73, 2008.

[29] Jansen, Bernhard and Ramasamy, Harigovind V. and Schunter, Matthias and Tanner, Axel. Architecting dependable and secure systems using virtualization. In Architecting Dependable Systems V, Springer-Verlag, pages 124-149, 2008.

Author Biographies

Fabrizio Baiardi graduated in Computer Science at Università di Pisa where is a Full Professor with Dipartimento di Informatica where he chairs one of the computer science degree. His main research interest in the computer security field is risk assessment and management of complex ICT infrastructures

Dario Maggiari graduated in Computer Science at Università di Pisa in 2008. Currently is a researcher in the field of virtual machines security and program static analysis.

Daniele Sgandurra graduated in Computer Science at Università di Pisa in 2006. He is currently a PhD student at the Department of Informatics at Università di Pisa, where his major research fields are virtual machines security, intrusion detection systems and operating systems security.