Modeling the Runtime Integrity of Cloud Servers: a Scoped Invariant Perspective

Jinpeng Wei
Florida International University
weijp@cs.fiu.edu

Calton Pu
Georgia Institute of Technology
calton@cc.gatech.edu

Carlos V. Rozas, Anand Rajan
Intel Corporation
{carlos.v.rozas,anand.rajan}@intel.com

Feng Zhu
Florida International University
fzhu001@fiu.edu

***Abstract*—One of the underpinnings of Cloud Computing security is the runtime integrity of individual Cloud servers. Due to the on-going discovery of runtime software vulnerabilities like buffer overflows, it is critical to be able to gauge the integrity of a Cloud server as it operates. In this paper, we propose scoped invariants as a primitive for analyzing the software system for its integrity properties. We report our experience with the modeling and detection of scoped invariants. The Xen Virtual Machine Manager is used for a case study. Our research detects a set of essential scoped invariants that are critical to the runtime integrity of Xen. One such property, that the addressable memory limit of a guest OS must not include Xen's code and data, is indispensable for Xen's guest isolation mechanism. The violation of this property demonstrates that the attacker only needs to modify a single byte in the Global Descriptor Table to achieve his goal.**

Key Words- integrity modeling; invariants detection; tools; Xen

I. INTRODUCTION

According to IDC's 2008 Cloud Services User Survey [13] of IT executives, security is the number one concern in adopting Cloud Computing. Part of the reason is that the operating systems supporting the Cloud are just the conventional ones used today, which means that they can be compromised and be infected with malware. Not surprisingly, a prospective Cloud user is concerned about delegating his data and computation to a Cloud server that can be compromised at runtime, even if the server starts in a known-good condition and the Cloud provider is trusted.

One way to relieve the concern of a potential Cloud user is remote attestation [19], which enables the Cloud user or a trusted third party to measure the "healthiness" (or integrity) of a Cloud server at runtime, so that the compromise (or degraded integrity) can be detected in a timely manner.

There has been a long line of research in software integrity ([1, 3, 7, 9, 10, 16, 19, 20, 21, 22, 23, and 28]), because malware like rootkits [20] must modify the victim software in some way, thus violating its integrity. In general, the integrity of a system can be approximated by a set of properties that must be satisfied by a "healthy" software system. For example, many rootkits modify the system call table, so a property evaluated by many integrity monitors is whether the system call table has known-good values. It is through such properties that an integrity monitor differentiates a "healthy" system from a corrupted one.

Identifying integrity properties is critical to the effectiveness of any integrity measurement *mechanism*, because without a good set of integrity properties, the use of such mechanisms can be severely limited. For example, if the integrity properties only cover system call table, a new rootkit can manipulate other function pointers (such as those found in device driver jump tables) to achieve its goal and remain undetected.

Therefore, in this paper we study the problem of systematically identifying integrity properties given the target software, which can then be used as input to an integrity measurement mechanism. Specifically, we make the following contributions:

We propose *scoped invariants* as an important class of integrity properties. Scoped invariants are code or data that has constant value under some context (called their scope). An example scoped invariant is the Interrupt Descriptor Table (IDT) entry for page fault, which contains a constant function pointer once the system finishes its initialization. Scoped invariants are building blocks of more general integrity properties, and are amenable to integrity checking.

Our second contribution is a dynamic analysis tool that detects scoped invariants. Our tool runs the target program in a machine emulator and monitors memory writes and events generated by the target program. Memory writes monitoring supports or rejects the hypothesis that a variable is an invariant, while event monitoring help decide the scopes in which hypotheses about invariants apply.

Our third contribution is a scoped invariants case study of the Xen Virtual Machine Manager [4], which is the foundational software of many Cloud providers. Our tool identifies 271 scoped invariants essential to Xen's runtime integrity, including one with GDT (Global Descriptor Table) that can be violated to defeat Xen's guest isolation mechanism.

The rest of the paper is organized as follows. Section 2 discusses the modeling of software integrity, and proposes scoped invariants as an important class of integrity properties. Section 3 presents an automated scoped invariants detection scheme based on dynamic monitoring and statistical inference. Section 4 discusses our implementation of an automated tool for deriving scoped invariants. Section 5 evaluates our methodology and tool by studying scoped invariants of Xen. Section 6 discusses related work, and Section 7 concludes the paper.

II. MODELING INTEGRITY USING SCOPED INVARIANTS

In this section, we first discuss the basics of integrity measurement and our assumptions; then we formally define *scoped invariants* as a class of integrity property.

A. Background on integrity measurement and security assumptions

An integrity measurement system typically consists of three components: the target system, the measurement agent, and the decision maker [19]. Our first assumption is that the measurement agent is isolated from and independent of the target system, therefore it has a true view of the internal states (including code and data) of the target system. This is a realistic assumption due to the popularity of machine emulators such as QEMU [5], and it has also been shown that the measurement agent can run on dedicated hardware such as a PCI card [20]. Our second assumption is that measurement results are securely stored and transferred to the decision maker. This can be supported by hardware such as a Trusted Platform Module (TPM) [27]. The third assumption is that the target system's states (e.g., code and data) may be compromised by a powerful adversary who can make arbitrary modifications; therefore the decision maker can rely on very few assumptions about the trustworthiness of the target system.

Based on these assumptions, the decision maker is given a true view of the target system, and its task is to estimate the "healthiness" of the target system. The healthiness include functional correctness (e.g., a function that is supposed to reduce the priority level of a task is not modified to actually increase the priority level), and non-functional correctness (e.g., the priority level can be modified by a privileged user instead of a normal user). In the following subsections, we model the healthiness as integrity properties.

Moreover, the healthiness of the target system may change over time, because it may be under constant attacks. Therefore, the integrity of the target system may need to be periodically reevaluated.

B. Formalizing integrity properties

In theory, any software system can be modeled as an automaton with states and state transitions. For simplicity of presentation, we assume that the system can be in one of *n* possible states: $s_1, s_2, ..., s_n$. Example states are initialization, entering a function, returning from a function, system termination, and so on. And each state is characterized by a particular combination of values of the system's internal variables. Based on this general formalization, we can model runtime software integrity as a set of properties $\{P_1(s), P_2(s), ..., P_m(s)\}$. A runtime property $P_i(s)$ is a function on state *s* that evaluates to *true* or *false*. If a system state *s* satisfies all P_i's, we can say that *s* is a "healthy" state. Different runtime properties may have different structures, but each of them can be generalized to be a Boolean expression with the operators $\wedge$(and), $\vee$(or), and $\neg$(not). More complex properties can be constructed out of primitive properties using the operators mentioned above. A primitive property has the form $\text{func}(v_1(s), v_2(s), ..., v_1(s))$ which takes variables $v_1(s), v_2(s), ..., v_1(s)$ and returns *true* or *false* (*v*(*s*) is the value of *v* in state *s*). *func* can have arithmetic operations inside as well as relationship operations like ==, <, and >.

C. Definition of scoped invariants

Scoped invariants are one special class of primitive property with the form: $v(t) == k, t \in [s_1, s_2)$. E.g., it stipulates that the value of variable *v* must be a specific value *k* when the system enters state s_1, and continue to be this value until the system enters another state s_2 (assuming that there is a sequence of state transitions from s_1 to s_2). We call such a primitive property a *scoped invariant*, and $[s_1, s_2)$ is called its *scope*. An example scoped invariant is a global variable whose value does not change after initialization (e.g., once the system enters the running state). For example, the Interrupt Descriptor Table (IDT) entry for page fault is such a scoped invariant. Scoped invariants can be regarded as a simplified form of temporal logic.

Scoped invariants represent an important class of integrity properties. They may include critical internal control data of the system (e.g., function addresses) that are supposed to remain constant. Examples of such scoped invariants include the Interrupt Descriptor Table (IDT), whose importance to system integrity has been well-understood. Another type of scoped invariant holds security policy data, and the violation of such invariants can directly defeat the corresponding security measures. For example, by tampering with the list of "bad" IP addresses, the attacker can defeat a blacklist-based IDS (Intrusion Detection System).

Note that the scopes of different invariants can vary significantly, depending on whether they are global variables, heap variables, or local variables. The scope of a global invariant can span as much as the entire execution of the program; the scope of a heap invariant must fall within the allocation and the freeing of the heap memory block; finally, the scope of an invariant that is a local variable in a function must be a subset of the interval between the entrance and the exit of the function.

In this paper, we focus on estimating the target system's integrity from the measurement of scoped invariants. Other forms of integrity properties are subjects of future research.

D. Using scoped invariants for integrity measurement: practical issues

Scoped invariants fit conveniently into the software integrity measurement paradigm because they are amenable to runtime attestation. Given a scoped invariant $v(t) == k, t \in [s_1, s_2)$, the measurement agent can start to read the value of variable *v* once the system enters state s_1. Then the decision maker can verify if the measurements of

v are "good" until the system enters state s_2. The verification of v is simple – just comparing the runtime measurements of v against some known-good value k. Note that k may be difficult to obtain if it depends on something external to the target program, e.g., configuration parameters. Here we assume that k has been determined somehow, e.g., using the dynamic detection technique discussed in Section III.

Although theoretically the definition of the scope of a scoped invariant is simple - just identifying the two boundary states, in a real system it is nontrivial, because typically we do not have an *explicit* and *direct* representation of program states. Instead, we can only *infer* program states from registers, main memory, or the file system. For example, if the program is sequential, the program counter (PC) can tell us the progress that has been made by the program since it is started. However, if there are loops in the program, PC *alone* may not be sufficient because the corresponding instruction may be part of a loop body and we do not know the number of iterations the program has gone through the loop body. In this case, we may need additional information such as the value of a *loop guard* variable to better infer the program state. Finally, when the program handles asynchronous events such as hardware interrupts, the program execution becomes non-deterministic and it may be very hard to reliably infer the program states.

Another related issue is the granularity of the program states, which influence the cost of integrity measurement. At one end of the spectrum, the program can have very coarse-grained states (e.g., *initialization*, *running*, and *termination*). Here the *running* state covers most of the program's life span. At the other end of the spectrum, the program can have very fine-grained states (e.g., one state per instruction execution or even multiple states within one instruction). While the most fine-grained states enable the integrity measurement agent to have the closest thus the clearest view of the target system, it is the most expensive. On the other hand, the coarse-grained states may lead the decision maker miss many important events (including integrity violations due to attacks), but it is cheaper for the decision maker to keep track of the program states. Therefore, there is a tradeoff between the granularity of program states and the effectiveness of integrity monitoring.

The third issue is the tracking of program states by the measurement agent. As we mentioned in Section A, an attacker may change the target program in arbitrary ways, so we cannot rely on the target program to notify the measurement agent about its states. Instead, we can only let the agent actively *poll* the state from a different domain. Specifically, the agent can run in a more privileged domain from which it can intercept the target program's execution and inspect registers, memory, and files of the target program. As will be discussed in Section III, a machine emulator is a good choice to run the measurement agent securely.

One related issue is performance overhead introduced by integrity measurement. As discussed above, a measurement agent needs to intercept the target program's execution, which causes delays in the target program. Obviously, the slowdown factor depends on the frequency (how often a measurement is taken) and duration (how long each measurement takes) of the measurements, and the duration depends on the number of invariants that need to be checked.

III. AUTOMATED DETECTION OF SCOPED INVARIANTS

A. Overview

The inference of scoped invariants can be labor-intensive and error-prone if performed manually. Therefore, tools are needed to automate this process.

By definition, a scoped invariant $v(t) == k, t \in [s_1, s_2)$ has a constant value k when the system state is between s_1 and s_2. Accordingly, the scoped invariant detection must answer the following questions for each scoped invariant: (1) what are the starting and end states that define the scope? (2) which variable (v) is involved? and (3) what is the known-good value (k)?

Note that scoped invariants are with respect to their scopes, i.e., the same variable can be an invariant in a narrower scope but not in a broader scope if the broader scope includes an operation that changes the value of the variable. Therefore, we must first decide the scope and then decide whether a variable is an invariant within that scope.

Our invariant detection employs a dynamic profiling approach. Specifically, we run the target program in a machine emulator and monitor memory writes and events generated by the target program. Memory writes monitoring supports or rejects the hypothesis that a variable is an invariant, while event monitoring help decide the scopes in which hypotheses about invariants apply. In the remainder of this section, we first discuss memory write monitoring, and then discuss event monitoring.

B. Memory writes monitoring

By definition, a scoped invariant should not be modified other than the initialization. In other words, a variable that is modified multiple times is unlikely an invariant. Based on this reasoning, we can detect invariants by observing how the target software modifies its variables: if a variable is modified multiple times, it is unlikely an invariant; otherwise, it is an invariant.

Using dynamic profiling, we run the target software and collect its modifications to variables, which translate to memory writes. There are multiple ways to do this, including program instrumentation and emulation. Using emulation, we can run the target software in a machine emulator, which can intercept every memory write operation (e.g., a MOV instruction). With this capability, we can record the target memory address and the value written in each memory write operation. The result of

dynamic profiling is a sequence of tuples: $w_1, w_2, \cdots, w_n$, where $w_i = (addr_i, v_i)$.

Given a sequence $w_1, w_2, \cdots, w_n$, we can compute the frequency c_i of updates to each unique address $addr_i$. Then, we can sort $addr_i$'s at the ascending order of c_i's, and the sorted list of $addr_i$'s is a list of potential invariants with the most likely at the beginning and the most unlikely at the end. Note that the computation here captures addresses that are updated at least once; addresses that are not updated in the sequence are automatically inserted at the beginning of the sorted list as the most likely invariants.

C. Event monitoring

In addition to memory writes, the machine emulator also intercepts other events that help define the scopes of the invariants. As discussed in Section D, program states can be defined at various granularities, with different tradeoff between integrity measurement precision and cost. We choose to monitor two types of such events: function calls and function returns. The reason is that functions can give semantic meaning for creating (by initialization) or re-creating (by updating) an invariant. In other words, we can say that the scope of an invariant is between when it gets its value in some function and when it is assigned a different value in another function. Tracking the invocations and returns from functions is thus important for determining the scopes of invariants.

For example, the global variable **opt_noirqbalance** of Xen controls whether IRQ balance should be enabled, and Xen allows this configuration parameter to be modified by the hypercall *platform_op*. Obviously, this variable is an invariant between two consecutive *platform_op* hypercalls that modify it.

IV. IMPLEMENTATION

We develop a prototype tool that can automatically derive invariants. As Fig. 1 shows, we first run the target software on top of QEMU [5], a CPU emulator, which enables us to log all memory write operations of the target software (by the MMU Arbitrator). We also log important system events such as entering and exiting a function, which represent program states that define invariant scopes. Then the Log Miner performs an offline processing of the log – given the sequence of memory write operations between two system events, ranking the memory locations based on the number of modifications to them (with the least modified on the top), and mapping the memory locations to global variables (using symbol information).

The output of the Log Miner is a list of candidate invariants, ranked from the most likely to the least likely. If a variable is indeed an invariant, it will be ranked high in the candidate list – e.g., we will not miss the true invariants. However, some non-invariant variables may be ranked high because the condition that leads to their updates is not satisfied during the limited profiling. This is a typical limitation of dynamic analysis, which can be remediated by profiling the target program multiple times each with a different set of input. We can also filter such non-invariant variables using static analysis of the source code, which is out of the scope of this paper.

V. EVALUATION

To test the applicability of scoped invariants, this section takes Xen as the target system to do several case studies. We first discuss the motivation of choosing Xen as the target system (Section A); next we discuss a scoped invariant with GDT which are critical to Xen's guest isolation mechanism (Section B). In Section C, we present the result of an automated study of Xen's global invariants.

A. Choice of Xen as the subject of study

Virtualization is the foundational technology for Cloud Computing, and Xen [4] is one representative VMM (virtual machine manager) that allows multiple operating systems (called guest OSes or simply guests) to share the same physical machine. As the lowest layer in the Cloud Computing software stack, the runtime integrity of Xen is the root of trust for a Cloud Computing environment.

It is generally believed that Xen is more secure than commodity operating systems such as Windows and Linux because it is smaller and simpler. However, we cannot rule out the possibility of a malicious modification to Xen at runtime. For example, There could be vulnerabilities with Xen that can be exploited [17, 18]. Even if Xen is completely bug-free, there are environmental issues such as DMA and system management mode (SMM) [12] that can modify Xen at runtime. Therefore we feel it useful to choose Xen as the target system to perform an integrity study. The particular Xen version studied in this paper is a pre-release of Xen 3.0.4.

B. Study of the GDT scoped invariant

One essential security goal of Xen is guest isolation, e.g., a guest operating system should not have access to information about other guests on the same platform, nor should a guest have access to Xen's internal state information.

This guest isolation goal is achieved by scoped invariants associated with some entries of the Global Descriptor Table (GDT) [11]. Specifically, to avoid unauthorized access to its internal state from guests, Xen leverages the standard IA-32 segmentation and protection rings architecture: a guest operating system runs in ring 1

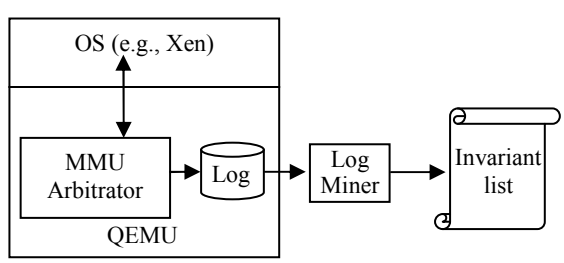

Figure 1. Scoped invariant detection Architecture

and guest processes run in ring 3, and four special *guest segments* are defined for them. For example, the data segment for ring 3 has the selector 0xe033 in the GDT. The "limit" of these guest segments is intentionally made smaller than 4GB such that Xen's code and data are excluded (Xen's code and data reside at the top of every address space).

Such a configuration is represented in the form of scoped invariants because information about these guest segments is stored in memory, in a data structure called **gdt_table**. Setting of the proper descriptor values for **gdt_table** is performed in the initialization phase of Xen, and after that the "limit" fields of the relevant entries are not supposed to change, in other words, they are scoped invariants.

It is easy to understand that a runtime modification to the **gdt_table** entries (e.g., setting the "limit" field to 4GB) could undo the effect of Xen's initialization and expose the complete 4 GB address space back to the guests. Then suddenly a guest can freely read Xen's data, violating the guest isolation security goal.

We have experimentally confirmed that modifying the "limit" field of the **gdt_table** entries at runtime enables a para-virtualized guest to read Xen's data and retrieve the list of domains on the platform by loading its **DS** register with 0xe033. This means that our hypothesis is valid. And it turns out that only one byte needs to be modified (from 0x67 to 0xFF). We should note that Xen virtualizes the GDT table for each guest domain, which means that each guest domain has its own GDT. However, each guest GDT derives its entries for the guest segments from the same **gdt_table**. Therefore, a modification to the **gdt_table** applies to all guest domains.

The GDT example demonstrates how a particular scoped invariant can influence Xen's high level security goals – i.e. guest isolation. Therefore, this invariant must be checked by a decision maker.

C. *A comprehensive detection of Xen scoped invariants*

We have performed a comprehensive study of scoped invariants for Xen, using the QEMU-based profiler and the Log Miner in Fig. 1.

We first ran Xen in the profiler, and used the Log Miner to generate the candidate scoped invariants list. Then we did a static analysis to confirm the real scoped invariants. Our static analysis scans the source code of Xen to locate all statements that write to a candidate invariant. We found that most of the candidate invariants have only one such statement (for initialization).

Our analysis suggests that most of the Xen global variables are scoped invariant at runtime. If we only consider the number of variables declared, 75% of them (271 out of 362) turn out to be invariants. If we also consider the size of the variables, then more than 90% of the memory locations corresponding to these global variables are invariant at runtime.

TABLE I shows some of the identified invariants. We have classified them based on an informal reasoning about why they should be invariants. Below we give details of some of these scoped invariants:

- **sched_sedf_def** is a data structure that stores the addresses of several functions that together implement the simple earliest deadline first (SEDF) algorithm of Xen. These functions are invoked when a virtual CPU is initialized, suspended, resumed, and so on. Obviously, they should be scoped invariants because otherwise an attacker can modify them to induce Xen's control flow to a malicious scheduling algorithm. Conceptually, **sched_sedf_def** is similar to the IDT. From TABLE I we can see that there are 27 more such scoped invariants in Xen.

TABLE I. SAMPLE SCOPED INVARIANTS (GLOBAL VARIABLES) IDENTIFIED FOR XEN

Type	Total Number	Examples
Static variables that are definitely invariants	63	schedulers, large_digits, small_digits
Effectively static structures (e.g., contains important function pointers)	28	sched_bvt_def, sched_sedf_def, ioapic_level_type, ioapic_edge_type, amd_mtrr_ops, apic_es7000, hvm_mmio_handlers, exception_table, hypercall_table
Variables that are effectively invariant given a particular boot configuration	17	opt_badpage, opt_sched, opt_conswitch, opt_console, acpi_param, debug_stack_lines, lowmem_emergency_pool_pages, dom0_nrpages
Variables that are effectively invariant given a hardware configuration	102	new_bios, ioapic_i8259, mp_bus_id_to_pci_bus, boot_cpu_logical_apicid, es7000_plat, dmi_ident, hpet_address, vmcs_size, max_cpus, max_page, cpu_present_map, vector_irq, irq_vector
Variables that are effectively invariant given a software configuration (e.g, functionality enabled)	4	softirq_handlers, gdt_table, change_point_list, key_table
Arrays whose entries are mostly invariant	7	idle_pg_table, idle_pg_table_l2, e820, e820_raw, irq_2_pin cpu_sibling_map, cpu_core_map, idt_table

TABLE II. MORE INFORMATION OF IDLE_PG_TABLE, IDLE_PG_TABLE_L2, AND IDT_TABLE

Table name	Start offset	Number of entries	Initialized By
idle_pg_table	0	4	xen/arch/x86/boot/x86_32.S
idle_pg_table_l2	DIRECTMAP_VIRT_START / (1<<L2_PAGETABLE_SHIFT)	DIRECTMAP_PHYS_END / (1<<L2_PAGETABLE_SHIFT)	__start in xen/arch/x86/boot/head.S
idle_pg_table_l2	0	16MB / (1<<L2_PAGETABLE_SHIFT)	__start in xen/arch/x86/boot/head.S
idle_pg_table_l2	FRAMETABLE_VIRT_START / (1<<L2_PAGETABLE_SHIFT)	(FRAMETABLE_MBYTES<<20) / (1<<L2_PAGETABLE_SHIFT)	init_frametable in xen/arch/x86/mm.c
idle_pg_table_l2	RDWR_MPT_VIRT_START >>L2_PAGETABLE_SHIFT	(max_page * BYTES_PER_LONG) >> L2_PAGETABLE_SHIFT	paging_init in xen/arch/x86/x86_32/mm.c
idle_pg_table_l2	RO_MPT_VIRT_START >> L2_PAGETABLE_SHIFT	(max_page * BYTES_PER_LONG) >> L2_PAGETABLE_SHIFT	paging_init in xen/arch/x86/x86_32/mm.c
idle_pg_table_l2	IOREMAP_VIRT_START >> L2_PAGETABLE_SHIFT	IOREMAP_MBYTES >> (L2_PAGETABLE_SHIFT – 20)	paging_init in xen/arch/x86/x86_32/mm.c
idt_table	0	128	init_IRQ in xen/arch/x86/i8259.c, apic_intr_init in xen/arch/x86/apic.c, trap_init in xen/arch/x86/traps.c, percpu_traps_init in xen/arch/x86/x86_32/traps.c
	129	127	
idt_table	128	1	dom0 kernel

- **opt_sched** holds the value of a boot-time parameter, which selects one of the built-in scheduling algorithms to be used by Xen. Since Xen does not support on-the-fly change of its scheduling algorithm, this variable should be a scoped invariant.

TABLE II gives more information about the invariants idle_pg_table, idle_pg_table_l2 and idt_table identified in TABLE I. First, since only part of such data structures (arrays) are invariants, TABLE II gives the range information. We have used macros (e.g., DIRECTMAP_VIRT_START) from Xen source code because their exact values depend on the hardware configuration (e.g., whether Physical Address Extension [11] is enabled). Second, the column denoted "Initialized By" shows the last function that sets the value of a particular scoped invariant. The goal of identifying functions in the "Initialized By" column is to specify the start of the scope of a scoped invariant, because since then the value of the scoped invariant is supposed to be constant.

D. Discussion

The degree to which a set of scoped invariants can approximate runtime integrity of a software system remains a research question. For example, the invariants that we identified are all necessary conditions, but they may not be sufficient. Assuming that a right set of scoped invariants is at hand, we can estimate the runtime integrity of the system by verifying them. If all of them are verified, we have more confidence about the system's integrity. But if some of them do not pass the verification, we know that the system has lost its integrity.

VI. RELATED WORK

Invariants detection

The Daikon invariant detector [8] generates likely invariants using program execution traces collected during sample runs. Daikon is the closest to our work in theory, but the two are different: Daikon instruments the program source code to emit data traces at specific program points, while our tool transparently intercepts program execution from a machine emulator.

Integrity measurement mechanisms

There has been a long line of research on integrity measurement. Approaches such as IMA [23] use hashing or digital signatures to measure the software at load time. Recently, ReDAS [15] and DynIMA [7] advance the state of the art by supporting software integrity measurement at runtime. Other related work includes [9, 19, 20, 21, 22, and 28]. These approaches generally focus on the mechanism for measurement, but not the integrity properties.

Copilot [20] is a co-processor based integrity checker for the Linux kernel. The properties that Copilot prototype checked were kernel code, module code, and jump tables of kernel function pointers. Although Copilot later provided a specification language [21], its focus was not on deriving integrity properties. We work out the properties from analyzing the target software itself.

Livewire [9] leverages a VMM (a modified version of VMware workstation) to implement a host-based intrusion detection system. It can inspect and monitor the states of a guest OS for detecting intrusions, and interposes on certain events, such as interrupts and updates to device and memory state. Like Copilot, Livewire does not focus on the identification of integrity properties but only checks known properties.

LKIM [19] produces detailed records of the states of security relevant structures within the Linux kernel using the concept of contextual inspection. However, the identification of security relevant structures relies on domain knowledge. This paper proposes an approach for systematically finding such structures.

Specialized integrity property measurement

Some specialized integrity properties have been measured, such as control flow integrity [1] and Information flow integrity [14]. [1] checks if the control transfer from one function to the next is consistent with a pre-computed control flow graph, so we can think of it as checking a sequence property of the target software. [14] checks the integrity of a system by reasoning about information flows. But it assumes that there is no direct memory modification attack, e.g., information flows are triggered by well-defined interfaces (function calls or file reads).

Rootkits detection and recovery

As we mentioned, there has been a lot of research on rootkits. A nice survey of rootkits and detection software is given in [20]. From [6] you can also find a list of popular rootkits. The integrity measurement mechanisms (such as [9, 20, 22, and 28]) mentioned above all can be used for rootkit detection. Some work such as [10] and [16] attempts to detect rootkits and recover the software from known-good copies.

Trusted computing

The Trusted Computing Group [26] has proposed several standards for measuring the integrity of a software system and storing the result in a TPM (Trusted Platform Module) [27] whose state cannot be corrupted by a potentially malicious host system. Industry vendors such as Intel have embedded TPM in their hardware. Such standards and technologies have provided the root of trust for secure booting [2], and enabled remote attestation [24]. There has been a consistent effort in building a small Trusted Computing Base (with hardware support such as TCG and application level technique such as AppCore [25]). A small Trusted Computing Base facilitates integrity analysis and monitoring.

VII. CONCLUSION

In this paper, we have studied the problem of modeling the runtime integrity of a Cloud server. We propose scoped invariants as an important class of integrity properties, and we design and implement automated tools that can derive scoped invariants out of the target software.

To evaluate our methodology, we apply our tools to the Xen VMM and identify 271 scoped invariants that are critical to Xen's runtime integrity. We experimentally confirm some of these invariants, including one that can be violated to defeat Xen's guest isolation mechanism.

VIII. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity", ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Nov. 2005.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture", In IEEE Computer Society Conference on Security and Privacy. IEEE, 1997, pp. 65–71.

[3] A. Baliga, P. Kamat and L. Iftode, "Lurking in the shadows: identifying systemic threats to kernel data", IEEE Symposium on Security and Privacy, Oakland, CA, May 2007.

[4] P. Barham, B. Dragovic, K. Fraser, et al., "Xen and the art of virtualization", ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, Oct. 2003.

[5] F. Bellard, "QEMU, a fast and portable dynamic translator", Proceedings of the 2005 USENIX Annual Technical Conference, 2005.

[6] chkrootkit. http://www.chkrootkit.org/, accessed August 16, 2010.

[7] L. Davi, A. Sadeghi, and M. Winandy, "Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks", Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC). November 2009.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants", In Science of Computer Programming, 2007.

[9] T. Garfinkel, M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection", Proceedings of Network and Distributed Systems Security Symposium (NDSS), February 2003.

[10] J. Grizzard, E. Dodson, G. Conti, J. Levine, and H. Owen, "Toward a trusted immutable kernel extension (TIKE) for self-healing systems: a virtual machine approach", Proceedings of 5th IEEE Information Assurance Workshop, 2004.

[11] Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A: System Programming Guide, Part 1.

[12] Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3B: System Programming Guide, Part 2.

[13] IT Cloud Services User Survey, pt.2: Top Benefits & Challenges. http://blogs.idc.com/ie/?p=210, accessed August 16, 2010.

[14] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture", Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006), June 2006.

[15] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence", Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), Lisbon, Portugal, 2009.

[16] J. Levine and J. Grizzard and H. Owen, "Re-establishing trust in compromised systems: recovering from rootkits that trojan the system call table", Proceedings of 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 2004.

[17] Xen local security-bypass vulnerability. http://www.securityfocus.com/bid/26954/discuss, accessed August 16, 2010.

[18] Xen "move-to-rr" RID local security bypass vulnerability. http://www.securityfocus.com/bid/26716/discuss, accessed August 16, 2010.

[19] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, C. D. McDonell, "Linux kernel integrity measurement using contextual inspection", Proceedings of the 2007 ACM workshop on Scalable Trusted Computing (STC). October 2007.

[20] N. Petroni, Jr., T. Fraser, J. Molina, W. A. Arbaugh, "Copilot—a coprocessor-based kernel runtime integrity monitor", 13th USENIX Security Symposium, San Diego, CA, Aug. 2004.

[21] N. Petroni, T. Fraser, A. Walters, and W. Arbaugh, "An architecture for specification-based detection of semantic integrity

violations in kernel dynamic data", 15th USENIX Security Symposium, 2006.
[22] N. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks", 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Oct. 2007.
[23] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture", 13th USENIX Security Symposium, 2004.
[24] J. Sheehy, G. Coker, J. Guttman, et al. "Attestation: evidence and trust", http://www.mitre.org/work/tech_papers/tech_papers_07/07_0186/07_0186.pdf, accessed August 16, 2010.
[25] L. Singaravelu, C. Pu, H. Haertig, C. Helmuth, "Reducing TCB complexity for security-sensitive applications: three case studies", 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, Leuven, Belgium, April 2006.
[26] Trusted Computing Group. http://www.trustedcomputinggroup.org, accessed August 16, 2010.
[27] Trusted Platform Modules. http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications, accessed August 16, 2010.
[28] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection", Tenth ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002.