Cloud Security Is Not (Just) Virtualization Security

A Short Paper

Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales
IBM T.J. Watson Research
{mihai,sailer,schales}@us.ibm.com

Daniele Sgandurra, Diego Zamboni
IBM Zurich Research
{dsg,dza}@zurich.ibm.com

ABSTRACT

Cloud infrastructure commonly relies on virtualization. Customers provide their own VMs, and the cloud provider runs them often without knowledge of the guest OSes or their configurations. However, cloud customers also want effective and efficient security for their VMs. Cloud providers offering security-as-a-service based on VM introspection promise the best of both worlds: efficient centralization and effective protection. Since customers can move images from one cloud to another, an effective solution requires learning what guest OS runs in each VM and securing the guest OS without relying on the guest OS functionality or an initially secure guest VM state.

We present a solution that is highly scalable in that it (i) centralizes guest protection into a security VM, (ii) supports Linux and Windows operating systems and can be easily extended to support new operating systems, (iii) does not assume any a-priori semantic knowledge of the guest, (iv) does not require any a-priori trust assumptions into any state of the guest VM. While other introspection monitoring solutions exist, to our knowledge none of them monitor guests on the semantic level required to effectively support both white- and black-listing of kernel functions, or allows to start monitoring VMs at any state during run-time, resumed from saved state, and cold-boot without the assumptions of a secure start state for monitoring.

Categories and Subject Descriptors

D.4 [**Security and Protection**]: Access controls

General Terms

Security

Keywords

integrity,outsourcing,virtualization,cloud computing

1. INTRODUCTION

Cloud computing holds significant promise to improve the deployment and management of services by allowing the efficient sharing of hardware resources. In a typical cloud scenario, a user uploads the code and data of their workload to a cloud provider, which in turn runs this workload without knowledge of its code internals or its configuration. Users benefits from offloading the management of their workload to the provider, while the provider gains from efficiently sharing their cloud infrastructure among workloads from multiple users. This sharing of execution environment together with the fact that the cloud user lacks control over the cloud infrastructure raises significant security concerns about the integrity and confidentiality of a user's workload. One underlying mechanism enabling cloud computing is virtualization, be it at the hardware, middleware, or application level. While a large amount of research has focused on improving the security of virtualized environments, our ongoing work on building virtualization-aware security mechanisms for the cloud has taught us that existing security techniques do not necessarily apply to the cloud because of the mismatch in security requirements and threat models.

In cloud computing, security applies to two layers in the software stack. First, users' workloads have to be run isolated from each other, so that one (malicious) user cannot affect or spy on another user's workload. Second, each user is also concerned with the security of their own workload, especially if it is exposed to the Internet (as in the case of a web service or Internet application). Many solutions exist for enforcing isolation between workloads, including the use of virtualization. Securing a particular workload is a much harder task and requires knowing what code is part of the workload. For example, in the case of an infrastructure cloud built on hardware virtualization, the workload is the guest operating system (OS) running in a virtual machine (VM). Because we have two parties involved (a cloud provider that controls the virtual-machine monitor (VMM) and a cloud user that controls the OS inside the VM), information about the guest OS is not readily available to the VMM. Securing the guest OS without relying on the guest OS functionality and without having *a priori* knowledge of the OS running in the guest VM falls outside the capabilities of existing solutions.

Current virtualization research assumes that the virtualization environment, e.g., the VMM, has knowledge of the software being virtualized, e.g., the guest OS. This knowledge can then be used to monitor the operation of the guest VM, to determine its integrity and to correct any observed anomalies. More specifically, all techniques proposing to monitor and enforce the security of an operating system inside a guest VM rely on several assumptions. First, the locations of code and data inside of the guest VM are often expected to be found based on symbol tables, with no verification of whether the memory layout of the running VM matches the symbol tables. Any malware that relocated security-sensitive data structures would fool detectors built on these techniques, and any valid update of the guest OS by the VM owner would result in numerous false alarms. Second, existing techniques need to monitor the

CCSW'09, November 13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-784-4/09/11 ...$10.00.

guest VM from the very moment when the guest OS boots. This is certainly unfeasible in infrastructure clouds, where the lifetime of a VM is decoupled from the lifetime of the guest OS running inside that VM (for example, a VM can start from a snapshot of the guest OS, in which case the VM starts with the OS fully loaded and running). These two assumptions make existing virtualization-based security techniques unsuitable for the cloud setting.

In this paper we describe the architecture we have developed to secure the customers' virtualized workloads in a cloud setting. Our solution, a secure version of virtual-machine introspection, makes no assumptions about the running state of the guest VM and no assumptions about its integrity. The OS in the guest VM is in an unknown state when our security mechanism is started, and we monitor it to discover its operation and measure its level of integrity. The monitoring infrastructure initially assumes only the integrity of the hardware state, as we presume that an attacker inside a VM cannot re-program the CPU. Starting from known hardware elements such as the interrupt descriptor table we explore automatically the code of the running VM, validating its integrity and that of the data structures on which it depends. This approach of combining the discovery of relevant code and data in the guest OS with the integrity measurements of the same code and data allows us to overcome the challenges of monitoring an a priori unknown guest OS without requiring a secure boot.

In this paper we make the following contributions:

- We introduce a new architecture for secure introspection, in which we combine discovery and integrity measurement of code and data starting from hardware state. Integrity measurements are done using whitelists of code executing in the VM, which need to be generated offline once for every supported operating system. This architecture addresses both the semantic gap present in virtual-machine introspection and the *information gap specific to cloud computing*, where no information about the software running in the guest VM is available outside the guest VM. Section 3 provides details of our security-oriented introspection mechanism.
- We present a technique to learn the exact type and version of an operating system running inside a guest VM. This technique builds on our secure-introspection infrastructure. Section 4.1 describes the technique and Section 5.1 evaluates its precision in comparison to existing OS-discovery tools from the forensic world.
- As a second application of our secure-introspection infrastructure, we design a rootkit-detection and -recovery service, which runs outside the guest VM and uses introspection to identify anomalous changes to guest-kernel data structures. When a rootkit is detected, it is rendered harmless by restoring the damaged kernel data structures to their valid state. Sections 4.2 and 5.2 respectively describe the design of this anti-rootkit service and a preliminary evaluation that highlights a high detection rate and lack of false positives.

2. RELATED WORK

Virtual Machine Introspection for security. *Virtual machine introspection* (VMI) was first proposed in [5] together with Livewire, a prototype IDS that uses VMI to monitor VMs. *XenAccess* [2] is a monitoring library for guest OS running on top Xen that applies VMI and virtual disk monitoring capabilities to access the memory state and disk activity of a target OS. Further VMI-based approaches are virtual machine replay [4] and detecting past intrusions [6]. These approaches mandate that the system is clean when it starts being monitored, which our solution VMs does not require.

Memory Protection. CoPilot [10] is a coprocessor-based kernel integrity monitor that performs checks on system memory to detect illegal modifications to a running Linux kernel. *Paladin* [1] is a framework that exploits virtualization to detect and contain rootkit attacks by leveraging the notion of *Memory Protected Zone* (MPZ) and *File Protected Zone* (FPZ) that are guarded against illegal accesses. For example, the memory image of the kernel and its jump tables are in the MPZ, which is set as non-writable. *XENKimono* [13] detects security policy violations on a kernel at run-time by checking the kernel from a distinct VM through VMI. It implements integrity checking, to detect illegal changes to kernel code and jump-tables, and cross-view detection and applies a whitelist-based mechanism, such as for checking the list of kernel modules that can be loaded into the kernel. *SecVisor* [16] is a tiny hypervisor that ensures that only user-approved code can execute in kernel mode. SecVisor virtualizes the physical memory, so that it can set hardware protections over kernel memory, and the CPU's MMU and the IOMMU to ensure that it can intercept and check all modifications to their states. These systems require information about the guest OS (e.g. location of data structures) to operate. Instead, our system only relies on hardware state and discovers any other information safely, before it is used.

Secure Code Execution. *Manitou* [7] is a system implemented within a VMM that ensures that a VM can only execute authorized code by computing the hash of each memory page before executing the code. Manitou sets the executable bit for the page only if the hash belongs to a list of authorized hashes. *NICKLE* [15] is a lightweight VMM-based system that transparently prevents unauthorized kernel code execution for unmodified commodity OSes by implementing *memory shadowing*: the VMM maintains a shadow physical memory for a running VM and it transparently routes at run-time guest kernel instruction fetches to the shadow memory so that it can guarantee that only the authenticated kernel code will be executed. One requirement for this system is that the guest OS is clean at boot and that is monitored continuously from power-on and throughout its life-cycle. With our framework, VMs can be created, cloned, reverted to snapshots and migrated arbitrarily throughout their lifetime.

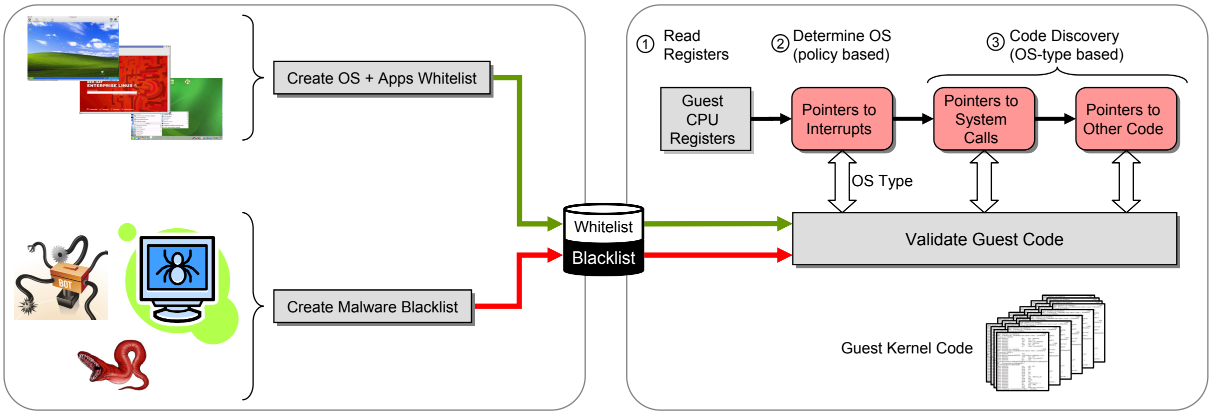
Secure Control Flow. *Lares* [9] is a framework that can control an application running in an untrusted guest VM by inserting protected hooks into the execution flow of a process to be monitored. These hooks transfer control to a security VM that checks the monitored application using VMI and security policies. Since the guest OS needs to be modified on the fly to insert hooks, this mechanism may not be applied to some customized OS. *KernelGuard* [14] is a prevention solution that blocks *dynamic data kernel rootkit attacks* by monitoring kernel memory access using a set of VMM policies. Petroni and Hicks [11] describe an approach to dynamically monitor kernel integrity based on a property called *state-based control-flow integrity*, which is composed of two steps: (i) validate kernel text, including all static control-flow transfers, by keeping a hash of the code; (ii) validate dynamic control-flow transfers. The runtime monitor considers the dynamic state of the kernel and then traverses it starting from a set of *roots* (kernel's global variables), and follows the pointers to locate each function that might be invoked and it verifies whether these pointers target valid code regions, according to the kernel's control flow graph. This system requires the kernel source code to apply static analysis to build the kernel's control flow graph, whereas in our solution we also check the integrity of OSes for which source code is not available.

3. OVERVIEW OF OUR ARCHITECTURE

Ensuring integrity in a running operating system is a daunting

Assumptions of existing work	Points of failure in a cloud environment
System is monitored continuously from power-on and throughout its lifecycle	VMs can be created, cloned, reverted to snapshots and migrated arbitrarily throughout their lifetime.
Guest system is clean when it starts being monitored	VMs can come into existence already infected or compromised.
Guest system can be modified on the fly to insert hooks or other monitoring mechanisms	Some customized systems may not be able to be modified, or we may not have the knowledge necessary to do the modifications.
Guest OS is known in advance	VMs may be configured with any one or more guest OSes (e.g., multi-boot VMs).
Guest OS source code is available	Most real-world attacks (e.g. rootkits) operate on Windows.
Guest OS information (e.g. location of data structures) is available	The location of internal data structures is unknown when no source code and no version information are available.
Malware is known in advance and given as blacklists	New malware is created constantly, realistic protection cannot rely on blacklists.
Trusted boot process exists	Not all h/w platforms, hypervisors, and OSes support trusted boot.

Table 1: Assumptions made by existing kernel integrity-checking mechanisms.



a) Lab Environment: One-Time Reference Measurements.

b) Cloud Environment: Run-time Checking of Guest VMs.

Figure 1: Overview of the integrity discovery system using secure introspection

challenge, and one that has been explored for a long time in the research community. In a system running on real hardware, all integrity checks need to be done from *within* the system being monitored, which inevitably raises the question of how to verify the integrity of the integrity-monitoring components themselves. Traditionally, this has been solved by requiring a trusted boot process to be in place, so that the integrity of the operating system and all its components can be verified by starting from power on.

With virtualization, it becomes possible to monitor the system "from the outside," through the use of virtual-machine introspection. This improves the situation by moving the monitoring components outside the monitored VM and outside the reach of an attacker. In addition to existing challenges of determining the code and data integrity of a running OS, building introspection-based monitors poses several new challenges:

- The *semantic gap* [3] between the level of detail observed by the monitor and the level of detail needed to make a security decision can only be bridged through deep knowledge of the guest OS.
- The complex lifecycle of VMs, which can be cloned, suspended, transferred, restarted, and modified arbitrarily makes any requirements for a trusted boot or for continuous monitoring unrealistic. The monitor must determine the integrity of the guest OS by starting only from the current state, without requiring any history.
- Operating systems for which source code is not available (in particular Windows, by far the largest target for rootkits and other malware) make bridging the semantic gap harder.

Table 1 presents some of the common assumptions made by existing kernel-integrity-protection work, and sample situations in which those assumptions break. These assumptions make most existing kernel-integrity-monitoring systems unable to protect VMs in a real production cloud environment.

Threat Model. In spite of these challenges, we wish to handle threats as generic as possible against cloud workloads. We allow *the attacker to control completely the guest virtual machines*, both the user-space applications and the operating-system kernel and associated drivers. Additionally, we assume that the cloud user (i.e., the victim) that owns these guest VMs does not provide the cloud provider with any information about the type, version, or patch status of the software running inside the guest VMs. We make only two assumptions. First, the hypervisor, which is under the control of the cloud provider, is correct, trusted, and cannot be breached. Second, there are virtual machines, again under the control of the cloud provider, which no attacker can breach. We will use these VMs (called *security VMs*) to host our discovery & integrity solution.

Our Secure-Introspection Technique. Figure 1(b) shows the steps to discover and verify the integrity of a guest kernel:

1. Read the IDT location from the virtual CPU registers.
2. Analyze the contents of the IDT, and using the hash values of in-memory code blocks and whitelists of known operating systems, determine the guest OS running in the VM.
3. Using the information about the running OS, use the appropriate algorithms to discover other operating system structures that are linked to from the IDT (e.g. system call tables, lists of processes and loaded kernel modules, etc.)
4. Continuously analyze all the discovered data structures using the whitelist appropriate for the guest OS, to determine when they are modified and if the modifications are authorized or not. Follow the execution of the code to the maximum extent possible to verify the integrity of as much of the kernel code as possible, during live execution of the guest VM.

The whitelists used by our approach consist of cryptographic hashes of normalized executable code found in the kernel (including modules and device drivers) of the operating system, plus some metadata to indicate the type and location of the entry. A whitelist needs to be produced for each supported OS type and version (or service pack), and for each whitelisted application, and can be generated offline (Figure 1(a)) using a clean installation of the OS, using an automated process. Blacklists are implemented by the same mechanism.

This algorithm addresses the problems mentioned in Table 1. It allows us to start monitoring a guest VM at any time in its life cycle and to monitor it correctly starting at that point, because the discovery of the OS structures depends only on the hardware state, which can be read at any moment. We can start monitoring a system that is already infected and we can correctly identify the infection, thanks to the use of whitelists. We do not need to know the guest OS in advance, since it is determined on the fly by the analysis, nor do we need access to the OS source code, making it particularly suitable for protecting against real-world attacks against both Windows and Linux. There is no need to know malicious code in advance. Thanks to the use of whitelists, any modifications to the guest OS will be correctly detected (and prevented, depending on policy). No trusted boot is required in the VMs. By assuming the hypervisor and the Security VM (from which the monitoring is done) are trusted, we can establish a "dynamic root of integrity" that allows us to dynamically determine the integrity of all critical components in the VM. Because secure introspection is non-intrusive, allowing us to examine the state of the virtual hardware in a completely transparent manner, no modifications need to be made to the guest system to support monitoring.

4. SECURE INTROSPECTION

To build the functionality required for secure introspection, we apply an iterative, incremental process of validating the integrity of kernel code and data. We assume the hardware to be trusted to perform as specified and to be impervious to attacks.[1] This assumption means that the hardware state which by specification defines control-flow transfers has values reflecting the true execution flow in the system. For example, if entry 0 of the interrupt description table (IDT) contains an interrupt-gate descriptor with the value `0xffffabcd`, then we know that the code at virtual address `0xffffabcd` will be invoked on a division-by-zero exception. Then we can bootstrap integrity by (a) validating all of the code pointed to by hardware state, then (b) identifying the kernel data used by the validated code, and (c) repeating the process for any code pointed to by the newly identified kernel data.

A pseudo-code sketch of our algorithm for secure introspection is given in Algorithm 1. Initially only the hardware "code" (i.e., the functionality of the hardware, including the microcode) is trusted. The algorithm relies on three subroutines. First, CFDataUsed returns the sets of hardware state and memory locations that influence the control flow out of a given code fragment. As a special case, CFDataUsed returns the hardware state used in hardware-mediated control-flow transfers (e.g., for Intel IA-32 processors this includes the `IDTR` register and the `msr_sysenter_cs`, `msr_sysenter_eip`, and `msr_star` model-specific registers). For other code, we derive their dependencies on memory locations *a priori*. In the case of indirect control transfers, for which we know the memory location or register used to direct the control flow but we do not know its actual value, we use execute triggers (via the introspection infrastructure) to gain control over the VM right before the control-flow transfer is about to occur. The second routine, CodeIsValid, computes a checksum over the code paths starting at the given location and checks it against a whitelist of known code checksums. Finally, MonitorForWrites simply monitors (via the introspection infrastructure) the memory regions occupied by the given code and data. When a write occurs to a monitored region, the corresponding code and data are scheduled for re-validation by removing them from the $Trusted$ sets.

Algorithm 1: Secure introspection

```
$TrustedCode \leftarrow \{hardware\}$ ;
$TrustedData \leftarrow \emptyset$ ;
while $true$ do
    $d \leftarrow \emptyset$ ;
    foreach $c \in TrustedCode$ do $d \leftarrow d \cup$ CFDataUsed($c$) ;
    $d \leftarrow d \setminus TrustedData$ ;
    foreach $ptr \in d$ do
        if CodeIsValid(code at $ptr$) then
            add code at $ptr$ to $TrustedCode$ ;
            add $ptr$ to $TrustedData$ ;
        else
            raise alarm ;
    MonitorForWrites($TrustedCode \cup TrustedData$) ;
end
```

This algorithm allows us to discover the integrity of the kernel code running inside the guest VM, without any expectations about the layout of that code. Because we follow the code paths throughout memory, we validate only the code that is actually run and do not need to worry about distinguishing between code and data on mixed-use memory pages.

4.1 Application #1: Guest-OS Identification

Asset identification and inventory is an important part of network and system management. The most common approach is to use network-based scanning to fingerprint devices connected to the network, using tools like nmap [8]. However, network-based fingerprinting can be easily defeated by programs running in the device, and this capability is widely available in programs like honeyd [12].

Using introspection to analyze the state and behavior of virtual machines provides advantages not only from the security point of view, but also from the system and network management point of view. One such advantage is the ability to precisely identify the operating system running in each VM, independently of the behavior of both user- and system-level programs in the VM. Through

[1] The problem of attacks that overwrite the BIOS, the firmware of various devices, or the processor microcode is outside the scope of this work.

experimentation, we established that the first code fragments that are validated in our secure-introspection algorithm are sufficient to uniquely identify the guest OS. In other words, the interrupt handlers (as pointed to by the IDT entries) vary significantly across OS types, versions, and even patch levels. It would be extremely difficult for an attacker to modify the interrupt handlers to fool the identification, while at the same time maintaining the guest OS in functioning state.

4.2 Application #2: Rootkit Detection

The secure-introspection algorithm provides information about the integrity of the kernel code, validating each code fragment present in memory against a whitelist of known code. Additionally, the validation procedure takes into account the control flow between code fragments, making sure that authorized code invokes only the authorized code using the appropriate control flows. Based on this functionality, we easily build a rootkit detector that works by identifying the presence of unauthorized code in kernel space. Every time the secure-introspection algorithm cannot validate a code fragment, it indicates that the kernel integrity might have been breached. Depending on the defined security policy, the secure-introspection monitor can raise alerts on all unknown code fragments, or can use a database of known malicious code (i.e., a blacklist) to reduce the number of false positives.

A novel feature we gain for free from secure introspection is the detection of rootkits (and more generally any malware) that disables security software present in the guest. Most security software hooks into kernel data structures that allow it to monitor security-sensitive operations such as file creation, file modification, or network communication. A malware can make such security software ineffective by unhooking its code from the kernel structures. Our introspection-based monitor observes this operation as a change of kernel pointer from one authorized code fragment to another authorized one. A simple security policy prevents such unhooking attacks by assigning priorities to authorized code, such that the code of a firewall handler takes precedence over the default code built into the OS.

5. EVALUATION

We looked to determine the effectiveness of our secure-introspection approach by evaluating two key metrics. First, we compared the accuracy of the OS-identification technique built on secure introspection with existing approaches. Second, we measured the detection and false-alarm rates of our rootkit detector that uses secure introspection.

To summarize our results, OS identification has perfect accuracy even where `nmap`-style techniques fail, and the anti-rootkit application has a high detection rate with no false positives. Overhead observed during experiments is minimal, with only few seconds of delay when the secure-introspection engine first connects to a guest VM and less than 2% overhead in macrobenchmarks. The experiments were performed on a 2.66GHz dual quad-core system with 18GB of memory, running 23 different guest OSes, including Microsoft Windows XP and 2003 at various service-pack levels, and multiple releases of RedHat, SuSE, and Ubuntu Linux, in both 32- and 64-bit versions. We used a commercial hypervisor with introspection capabilities.

5.1 Guest-OS Identification

To test this capability, we set up a test network using honeyd in a Linux VM, and used both nmap and our approach to identify the operating system running on it. Honeyd is set up to simulate two devices: a generic Windows machine with some services (POP, HTTP, FTP, SSH) and a Cisco router with Telnet enabled. Figure 2 shows the experimental setup.

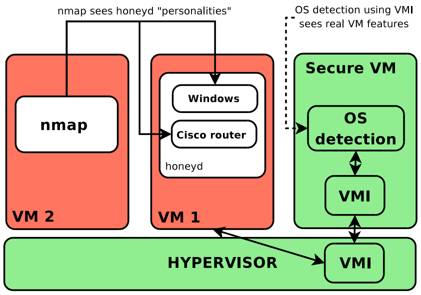

nmap sees honeyd "personalities"
OS detection using VMI sees real VM features
nmap
VM 2
Windows
Cisco router
honeyd
VM 1
Secure VM
OS detection
VMI
VMI
HYPERVISOR

Figure 2: Experimental setup for guest-OS identification.

```
Starting Nmap 4.62 ( http://nmap.org ) at ...
Interesting ports on 192.168.1.1:
PORT     STATE    SERVICE
21/tcp   open     ftp
22/tcp   open     ssh
25/tcp   filtered smtp
80/tcp   open     http
110/tcp  open     pop3
Device type: general purpose
Running (JUST GUESSING) :
    Microsoft Windows NT|95|98 (91%)
Aggressive OS guesses:
    Microsoft Windows NT 4.0 SP5 - SP6 (91%),
    Microsoft Windows 95 (90%), ...
No exact OS matches for host.

Interesting ports on 192.168.1.150:
PORT   STATE SERVICE
23/tcp open  telnet
Aggressive OS guesses:
    Vegastream Vega 400 VoIP Gateway (91%),
    D-Link DPR-1260 print server,
    or DGL-4300 or DIR-655 router (90%), ...
No exact OS matches for host
```

Figure 3: Nmap run on honeyd hosts (redacted for space).

```
Initializing Introspection Manager...
...
waiting for VM to be attached
VM is attached to agent 192.168.1.34:8080
CPU started.
Operating System Detection active.
Reboot Monitoring active.
Guest OS identified as Linux.
guest OS reported: Linux/RHEL Linux 5.2
  (32-bit)/32-bit/SMP/0.0.0.0
```

Figure 4: OS detection using secure introspection.

We ran the experiment by running nmap from a different VM on both honeyd addresses, at the same time as our OS identification code was running on the SVM. Nmap identified the honeyd “personalities” as Windows (with a confidence of 91%) and as different network devices, respectively. Our code identifed the VM correctly as Linux. These results are shown in Figures 3 and 4, respectively.

5.2 Rootkit Detection

As an example we use the Trojan W32/Haxdoor.AU, which infects Windows systems. Upon execution, the trojan drops some files into the Windows System folder (among others: `ycsvgd.sys`), and hooks several services in the System Service Descriptor Table (SSDT) to hide itself and to inject code into the `explorer`

```
[1] Event source: ARK Engine 1, Pid 32369
[2] Type of event: SSDT, Entry 173
[3] Driver: \??\C:\WINDOWS\system32\ycsvgd.sys
[4] Owner: W32/Haxdoor.AU
[5] ControlFlowHash: [SHA256 hash]
[6] Severity: High
[7] Action: Monitor
```

Figure 5: Information provided in a rootkit-detection event.

`.exe` process, code which it later executes as a remote thread. When activating W32/Haxdoor.AU on our monitored 32-bit Windows XP SP2 VM, the anti-rootkit shows changes to six system services (`NtQueryDirectoryFile`, `NtOpenProcess`, `NtQuerySystemInformation`, `NtCreateProcessEx`, `NtCreateProcess`, and `NtOpenThread`). We show in Figure 5 the event generated by the anti-rootkit when the `NtQuerySystemInformation` service entry (SSDT Entry #173) is manipulated to illustrate the information it yields. In this example, the entry is redirected to point into rootkit code mapped from file `ycsvgd.sys`.

Monitoring the kernel control flow semantically enables the anti-rootkit engine also to ensure that the routines on which firewalls and antivirus software rely to inspect files and traffic are active and unaltered. We can detect and alert if a rootkit (cf. Unhooker) succeeds in unhooking these routines, rendering the running AV or firewall ineffective. Additionally, the anti-rootkit engine can also restore the hooks used by the firewall or the AV software, taking care to perform this step only if the corresponding firewall or AV code is still present in memory.

We focused our current implementation of the anti-rootkit engine to kernel-level malware, as monitoring the user space of the guest VM imposes an excessive overhead. Too many events would need to be monitored, leading to many expensive context switches to and from the Security VM. We plan to address this limitation through the injection into the guest VM of security agents, which would then run locally to identify user-space malware.

5.3 Performance

To measure the performance of the introspection, we performed two category of benchmarks. In the first, we measured the performance of the introspection layer. In particular, we measured the rate at which memory could be copied from the monitored guest to the SVM. In our setting, we could copy 2500 pages/second, which is well above the 2.8 memory pages inspected on average by the secure-introspection technique during the steady state of the guest OS. When the monitor first connects to a guest VM, the number of pages initially retrieved can reach 200.

For the second test, we measured the actual impact that the monitoring had on performance of the guest. `httperf` was used to assess the performance of a web server running on the guest with monitoring enabled and disabled. Overhead from monitoring using periodic (one second) checks was less than 2%.

6. CONCLUSIONS AND FUTURE WORK

While clouds are moving workloads closer together to save energy by better utilizing hardware, they also depend on reliable malware detection and immediate intrusion response to mitigate the impact of malicious guests on closely co-located peers. In this work, we have described how we securely bridged the semantic gap into the operating system semantics. The presented solution enables novel security services for fast changing cloud environments where customers run a variety of guest operating systems, which need to be monitored closely and quarantined promptly in case of compromise.

We are currently extending our framework with a mechanism to transparently inject a *context agent* from a Security VM into guest VMs through the introspection interface. While it has to be protected through introspection by a Security VM, the context agent can bridge the semantic gap by providing the Security VM with high-level information about the guest VM, such as the list of the running processes, open files and network connections, logged users, running kernel modules and so on. Agent injection holds the promise of bridging the semantic gap to any level of detail desired while eliminating most of the monitoring overhead.

7. REFERENCES

[1] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. *Department of Computer Science, Rutgers University, April.*, 2006.

[2] Bryan D. Payne and Martim Carbone and Wenke Lee. Secure and flexible monitoring of virtual machines. *Computer Security Applications Conference, Annual*, 0:385–397, 2007.

[3] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[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] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Symposium*, 2003.

[6] 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*, pages 91–104, New York, NY, USA, 2005. ACM.

[7] 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.

[8] G. F. Lyon. *NMAP Network Scanning*. Nmap Project, 2009.

[9] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. *Security and Privacy, IEEE Symposium on*, 0:233–247, 2008.

[10] N. L. Petroni, Jr., 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*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[11] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, New York, NY, USA, 2007. ACM.

[12] N. Provos. Honeyd — A virtual honeypot daemon. In *10th DFN-CERT Workshop,*, Hamburg, Germany, Feb. 2003.

[13] 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.

[14] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring. In *Proceedings of ARES 2009 Conference*, 2009. To appear.

[15] 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*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] 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*, pages 335–350, New York, NY, USA, 2007. ACM.