Computer Meteorology: Monitoring Compute Clouds

Lionel Litty H. Andrés Lagar-Cavilla
Dept. of Computer Science
University of Toronto

David Lie
Dept. of Elec. and Comp. Eng.
University of Toronto

Abstract

Cloud computing environments allow customers to execute arbitrary code on hardware owned by a cloud provider. While cloud providers use virtualization to ensure isolation between customers, they face additional security challenges. Malicious customers may leverage the provider's hardware to launch attacks, either from VMs they own or by compromising VMs from benign customers. These attacks can damage the provider's reputation and ability to serve other customers. In this paper we show that while cloud providers can use introspection to monitor customer VMs and detect malicious activity, it must be used with care since existing introspection techniques are based on assumptions that do not hold in cloud environments.

1 Introduction

A virtual machine monitor (VMM) decouples OS images from the hardware they run on, enabling useful capabilities such as moving virtual machines (VM) between hosts, consolidating several underutilized VMs onto a single host, and the ability to checkpoint/rollback VMs. The availability of such capabilities has lowered data center costs immensely. Recently, it has also made possible one of the many forms of *cloud computing*, sometimes referred to as *Infrastructure as a Service*, in which customers execute their OS images on hardware rented from *cloud providers* such as Amazon's EC2 service, GoGrid or Mosso. By shifting the burden of infrastructure ownership and maintenance to the provider, cloud computing allows subscribers to scale their applications and leverage large pools of resources while only covering costs proportional to their actual resource usage.

While enticing, there remain significant obstacles to this vision. In a recent survey of 244 IT executives and CIOs, security was ranked as the number one challenge facing cloud computing [5]. Cloud computing introduces unique security challenges for cloud providers. Most importantly, the provider needs to guarantee isolation between customers. This means that a malicious customer should not only be unable to access information about other customers, she should also be unable to affect the performance of other customers. This is the guarantee virtual machine monitors aim to provide.

While cloud providers can rely on a properly designed virtual machine monitor to isolate VMs from one another, they also have to adopt some form of cloud monitoring in order to prevent their infrastructure from becoming a haven for malicious activities. In addition to hurting the provider's reputation – and consequently its business, malicious activities occurring on its cloud have technical consequences. For example, spam e-mails were discovered to have originated from IP addresses belonging to Amazon's EC2 service, which resulted in the blacklisting of a large swath of Amazon's IP addresses [7]. The IP addresses of the cloud provider are a shared resource that can permit one misbehaving customer to adversely affect other customers. The terms of service of the aforementioned cloud providers have broad language prohibiting illegal activities [1, 4], but the technical means that can be used to enforce these terms of services are currently ill-defined. For example, a cloud provider could implement network-level monitoring and control in a way similar to that of an Internet Service Provider (ISP). While this may detect some malicious activities, it is not a panacea. To wit, deep inspection of encrypted traffic is not possible. Moreover, even unencrypted malicious traffic can be made challenging to detect via network monitoring – to hide a network scan an attacker may use a botnet to perform a stealthy, distributed scan.

For these reasons, ISPs have had, for the most part, a great deal of difficulty protecting their networks from abuse using only network monitoring. However, cloud providers have the advantage of being able to use *introspection* to assist them in analyzing VMs. In this paper, we explore to which extent VM introspection techniques can be used to monitor cloud customer VMs for signs of misbehavior. While introspection has been extensively studied in the literature [2, 3, 6, 8, 9], previous introspection systems assumed that the guest VM and the VMM were owned by the same principal, making it easier for the administrator to use knowledge they have about the guest VMs to tune VM introspection. This is not the case for a cloud provider, who would ideally like to place as few restrictions on the customer VMs as possible. The cloud provider faces a significantly different introspection problem because it can make very few assumptions about the operating system version and configuration of its customer's VMs. As a result, a cloud provider must be very conservative about how it interprets its monitoring state, for fear of incorrectly labeling a benign customer as malicious.

To help map research opportunities, we propose a taxonomy of existing introspection techniques and examine their limitations. We propose tamper-evident, architectural monitoring as a first step towards improving the applicability of introspection to the cloud setting. However, other solutions exhibiting the same characteristics are needed to provide a complete solution. Failing that, introspection will be of limited use to cloud providers and they will have to primarily resort to network monitoring for their monitoring needs.

2 Introspection

A VM typically contains an enormous amount of code and data. To make introspection tractable, it makes sense to adopt a reductionist approach that seeks to divide a VM into smaller, manageable components and analyze them individually. Since a running VM can naturally be conceived of as a set of processes collaborating to perform tasks, the process boundary is a natural way of compartmentalizing events in a VM. This approach also follows naturally from the OS's attempt to isolate untrusting principals into separate processes. We will not discuss here reverse-engineering running code to try to determine its intent. Instead, we envision that the cloud provider will rely on establishing the identity of the code running in a process (e.g., Apache or MySQL) to formulate policies. Having identified individual processes, introspection can then be used to observe unencrypted traffic in the memory of the process. Likewise, distributed port scanning can be detected by observing that known malicious software is executing in a customer VM.

The capabilities of introspection are determined by the nature of the VMs being monitored. In this paper, we distinguish between the two types of VMs: *Non-malicious VMs* may be vulnerable to attacks, but have not yet been compromised. *Malicious VMs* may either belong to malicious customers or be originally non-malicious VMs that have been taken over by a malicious attacker. While the idea of a VM as a set of distinct processes that can be analyzed separately is attractive, a malicious VM can easily blur the boundaries between processes, defeating a reductionist approach. Attackers who gain administrative privileges in a commodity OS are usually able to circumvent the process-level protections implemented by the OS. For example, an attacker can use the `ptrace` debugging facility on Linux to manipulate the memory and alter the behavior of another process. Nearly every other operating system (including Windows) provides such powers to an attacker who gains administrative control. As a result, a reductionist approach is only applicable to non-malicious VMs and malicious VMs where the attacker is able to exert only limited control over a single process. However, before an attacker fully compromises a VM, they initially control only a single process, affording the cloud provider a window of opportunity to detect that the VM has transitioned from non-malicious to malicious. This window will close once the attacker has gained sufficient control of the VM to defeat monitoring.

Ideally, monitors that use introspection should be able to detect this transition from a benign to a malicious state. To do this, we require them to be *tamper-evident*, meaning that the monitor will either report complete and accurate information, or it will report that the information is incomplete. This means that an attacker controlling the VM cannot fool a tamper-evident monitor into reporting misleading information. A tamper-evident monitor is thus useful in its capability to detect transitions to a potentially malicious state, and in its ability to not be mislead by an attack. Monitors that are not tamper-evident are still useful: they report accurate information for non-malicious VMs. In addition, they may also report accurate information for compromised VMs when the attacker is unaware of the existence of the monitors, or when the attacker has only gained partial control of the VM being monitored.

3 Introspection Approaches

Several approaches can be used to bridge the semantic gap [2] and monitor the activity of cloud VMs. Below, we examine four representative introspection approaches and their limitations: host-based agent, trap and inspect, checkpoint and rollback, and architectural monitoring. We examine these approaches along three axes: power, unintrusiveness and robustness.

We define the *power* of an approach as the scope of VM events it can monitor as well as its ability to interpose on specific events. The *unintrusiveness* of an approach characterizes how much disturbance it introduces in the monitored VM. Lastly, the *robustness* of an ap-

	Power	Unintrusive-ness	Robust-ness
Host agent	Good	Poor	Good
Host agent w/ driver	Best	Worst	Poor
Trap/Inspect	Best	Good	Worst
Checkpoint/Rollback	Best	Good	Poor
Architectural	Poor(?)	Good	Best

Table 1: Capabilities of introspection techniques

proach depends on the nature of the assumptions made about the monitored VMs and how likely these assumptions are to hold. This includes robustness to different versions of an OS or even different OSs, as well as robustness to OS modifications as a result of an update, the introduction of new code in the form of a module or even malicious changes aimed at evading the monitoring. Table 1 summarizes the techniques studied in this section in terms of these three properties.

3.1 Host-based Agent

A host-based agent is an application that runs within the context of the monitored VM, either in user-space or as a kernel module. The cloud provider can ask customers to install the agent on all their VMs. Alternatively, the provider can inject that agent in any VM that is instantiated on the cloud. Since it runs in the VM context, the agent can use the OS API to acquire information about the VM. Furthermore, if the OS provides hooks to be notified upon certain events, the agent can leverage them to monitor changes in the VM. An agent can even extend the kernel by adding its own hooks.

This is the most intrusive approach. It breaks the boundary separating the provider realm from the customer realm, thus creating undesirable coupling between the two. Asking the customer to install an agent is burdensome and requires cooperation from a potentially incompetent or undisciplined customer. It also hampers cloud interoperability because it forces the customer to install an agent for each cloud provider it uses. The alternative, injecting an agent in an unknown VM, is unreliable. If the agent is injected in user space, its operation could be hampered by access control mechanisms of the customer VM, such as SELinux. Worse, it could trigger the customer's own security monitoring. Injecting the code in kernel space may ensure that the agent is not hampered by OS security mechanisms but adding code to a running kernel (and removing it when the VM leaves the cloud) is a daunting task.

Robustness and power depend on the design of the agent and the features provided by the OS. Different OSs are unlikely to provide compatible APIs for system monitoring. Furthermore, if the API provided is not sufficiently powerful, the agent will have to extend it by adding kernel code via a driver or module. This will increase the power of the approach at the cost of robustness, potentially coupling the agent to a specific version of the OS. Finally, the agent has to trust the OS to isolate it from other applications.

3.2 Trap and Inspect

Trap and inspect is a less intrusive approach that consists of examining the execution of a VM from the VMM or from another VM with the help of the VMM. Compared to an agent, it has the advantage of isolating the introspection code from the introspected VM, preventing tampering with the code and avoiding interference with the execution context of the VM. It also does not interfere with cloud interoperability, since it does not modify the VM. With adequate support from the VMM, introspection code is a powerful approach that is afforded full visibility over the introspected VM. It can observe hardware events such as disk accesses and network accesses, as well as the content of memory, making it as powerful as a host agent with a kernel component.

However, to be alerted on specific events, the monitor needs to insert traps in the VM code, much like a debugger. And like a debugger, this requires additional information to locate which instructions in code memory should be replaced by traps, as well as a thorough understanding of the code. Because this trap and inspect depends on knowing minute details about the code being monitored, it is the least robust of all introspection techniques. Trap placement for access control has been shown to be a complex problem that is hard to get right [10, 11]. Moreover, bridging the semantic gap requires inspecting data structures in memory, another task that requires expert knowledge of the OS used by the VM. The complexity of trap and inspect makes it brittle in the face of skilled attackers, as we will show in Section 4.1. As a result, trap and inspect is an engineering heavy task that is closely tied to a specific version of an operating system, resulting in an approach with low robustness.

3.3 Checkpoint and Rollback

To alleviate the need to understand the memory layout of the data structures of the OS of the monitored VM, checkpoint and roll back takes advantage of the VMM's ability to perform these operations on VMs. The introspection monitor can checkpoint a VM, inject code that can call any support function provided by the OS of the monitored VM and then roll back the VM to the checkpoint [6]. As a result, this approach subsumes trap and inspect: it is similarly powerful while being more robust, since it can leverage OS APIs to perform queries as opposed to reverse-engineer in-memory data structures. However, checkpoint and roll back still relies on traps to be invoked.

3.4 Architectural Introspection

Architectural introspection is a new approach first proposed in [8] to increase the robustness of the previous approaches in two ways. The principle behind architectural introspection is to restrict monitoring to only well-defined interfaces that are difficult or unlikely to change. In this way, architectural monitoring achieves both robustness, because the interfaces are stable, and tamper-evidence, because an attacker will have a difficult time changing that interface. Architectural introspection is both robust and unintrusive, since it only monitors stable, low-level interfaces through the VMM. For example, it may rely on interfaces such as the processor instruction set, MMU protection, executable file formats and file system layouts. These interfaces are either OS-independent or OS version-independent, a property we call *OS-agnostic*. Because monitoring is achieved by making only minimal assumptions about the OS, architectural introspection is the only introspection technique that can be tamper-evident as well. Architectural monitoring does not insert traps into the OS image, but instead relies on passively monitoring hardware events. As a result, this suggests that architectural monitoring will have weaker capabilities than other monitoring methods. Nevertheless, we believe that the power of architectural monitoring can be sufficient for cloud monitoring. We will give some examples of architectural introspection in Section 4.

4 Introspection Example

In this section, we give an example of an introspection task and discuss the trade-offs between the introspection techniques. The introspection task we examine is determining which applications are being run by a customer VM, including which version of the application is being run. For example, a cloud provider could use this knowledge to determine whether a customer is running a vulnerable version of network facing software. The cloud provider could also deploy vulnerability specific network filters to protect these vulnerable applications until they are patched, which the cloud provider can also detect. Additionally, the provider could isolate VMs running only known good software with no publicly known vulnerability from VMs running known vulnerable, malicious, or/and unknown software, giving their customers stronger privacy guarantees.

4.1 Execution monitoring

We begin by examining how introspection can be used to identify all running binary code in a VM. This capability is already natively provided by many operating systems. For example, the Linux kernel reports the code loaded in the address space of each process in `/proc`. Likewise, tools such as Process Explorer can provide the same information in Windows. Requiring the customer to install such tools and querying them from the VMM is an example of host-based agent monitoring. Note that Process Explorer requires a driver to be loaded in the Windows kernel, meaning that it belongs to the host agent with driver category. Numerous rootkits that hide the existence of entire processes have demonstrated that this approach is not tamper-evident.

Trap and inspect approaches also exist for execution monitoring [9]. As discussed above, these approaches require expert knowledge of the monitored OS. Similar to host-based monitoring, an attacker can circumvent trap and inspect monitors. Trap and inspect relies on the execution of instrumented instructions to invoke the monitor. If the attacker can insert code in the kernel (by loading a module for example), she can create new code paths that will not be monitored. We note that checkpoint and rollback introspection will be equally vulnerable to such an attacker since it also requires the execution of trap instructions to invoke the monitor.

In the case of execution monitoring, we have shown in previous work that architectural introspection can be as powerful as other introspection techniques [8]. Patagonix uses the processor MMU to receive notifications whenever binary code is executed, and identifies the code using the binary format specification. The Patagonix monitor is tamper-evident, because it never misidentifies code. If there is code that it does not recognize, either because it is malicious or because it is in a form that it cannot understand, it will report it as unidentified. The approach is OS-agnostic: any OS can be monitored, and if this OS uses an executable file format understood by the monitor, the executed code will be identified.

4.2 File monitoring

Execution monitoring only tells the cloud provider what binaries are executing, but does not provide any information about interpreted or dynamically generated code. For instance, Java may be used to run Eclipse or Tomcat, and the PHP interpreter may be used to run phpBB. Once a process has been identified as an interpreter or JIT, we want to determine which files it has read to allow decision making based on the scripts or byte-code it is executing.

Just as with execution monitoring, a variety of host based introspection solutions already exist and are supported by most OSs. Examples of host-based agents providing these features include `filemon` on Windows, `strace` on Linux. Trap and inspect, as well as checkpoint and rollback can also be used to obtain this information. The difficulty with these techniques arises, just as with execution monitoring, when the VM is malicious or running an unknown OS.

We are currently exploring methods by which architectural introspection can be used to identify files that are accessed by interpreters. The key idea is to interpret data on disk using the file system specification to identify blocks that belong to files of interest (for example a vulnerable JAVA class file). If information from such files flows into the address space of a binary of interest (for example a JAVA virtual machine), then we can detect the misuse of a file. To make such information flow tracking OS-agnostic, we again leverage the processor MMU and virtual DMA engine to track when the block is read off disk and into the buffer cache, and again subsequently when the data is read from the buffer cache and copied into the process address space. Mapping file accesses to specific processes is essential because sensitive files may be accessed by processes that are not interpreters, e.g., anti-viruses or file indexers. Flagging all accesses to a sensitive file would result in false positives. Our current prototype relies on two assumptions about the OS. First, that data is only copied out of the buffer cache when the process accessing the file is in context. Second, that data from the disk is accessed via page sized blocks using DMA.

5 Conclusion

In conclusion, we can now give the trade-offs between the various introspection techniques that a cloud provider can use to monitor their cloud. It is clear that architectural introspection has the best properties in terms of unintrusiveness and robustness. By design, architectural introspection allows the monitoring of a wide range of OSs, an important concern in the cloud. In addition, because it makes the fewest assumptions about the VM and its OS, it is the easiest technique to make tamper-evident. The main drawback with OS-agnostic techniques is that because architectural introspection restricts itself to monitoring at fixed interfaces, there may be cases where it is less powerful than other techniques because it cannot make implementation-specific assumptions. Thus, so long as architectural introspection reports enough information for the cloud provider's needs, it is the obviously best choice.

In cases where architectural introspection is insufficient, the cloud provider faces a trade-off between robustness and unintrusiveness. If the customer can be trusted to install the provider's monitoring agent in each of her VMs, then the agent will provide a more robust solution, especially if the agent can operate using solely APIs provided by the OS. Otherwise, when requiring the installation of an agent is too burdensome, the cloud provider must fall back to using checkpoint and rollback.

Finally, even if the cloud provider can tolerate the lowered robustness that comes with checkpoint and rollback, or the provider's customers can tolerate the installation of a monitoring agent, these introspection solutions must still be coupled with an architectural approach since they are not tamper evident-and may fail silently if the customer's VM is or becomes malicious.

Acknowledgments

We would like to acknowledge comments from James Huang, Lee Chew, Stan Kvasov, Vladan Djeric and Ashvin Goel, which helped improve this paper. Lionel Litty is supported by funding from the ISSNet Strategic Network and Bell University Labs.

References

[1] Amazon. Amazon web services customer agreement, 2009. `http://aws.amazon.com/agreement/` Last accessed: 4/2/2009.

[2] P. M. Chen and B. D. Noble. When virtual is better than real. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[3] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *10th Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, Feb. 2003.

[4] GoGrid. Gogrid cloud hosting: Acceptable use policy, 2009. `http://www.gogrid.com/legal/aup.php` Last accessed: 4/2/2009.

[5] IDC Exchange. IT cloud services user survey, pt.2: Top benefits & challenges, Oct. 2008. `http://blogs.idc.com/ie/?p210` Last accessed: 1/10/2009.

[6] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Oct. 2005.

[7] B. Krebs. Amazon: Hey spammers, get off my cloud!, July 2008. `http://blog.washingtonpost.com/securityfix/2008/07/amazon_hey_spammers_get_off_my.html` Last accessed: 1/10/2009.

[8] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *17th USENIX Security Symposium*, pages 243–258, July 2008.

[9] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, pages 289–304, July 2006.

[10] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *17th USENIX Security Symposium*, pages 379–394, July 2008.

[11] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *11th USENIX Security Symposium*, pages 33–48, Aug. 2002.