CloudPhylactor: Harnessing Mandatory Access Control for Virtual Machine Introspection in Cloud Data Centers

Benjamin Taubmann, Noëlle Rakotondravony, Hans P. Reiser
University of Passau
{bt, nr, hr}@sec.uni-passau.de

***Abstract*—Virtual machine introspection is a valuable approach for malware analysis and forensic evidence collection on virtual machines. However, there are no feasible solutions how it can be used in production systems of cloud providers. In this paper, we present the CloudPhylactor architecture. It harnesses the mandatory access control of Xen to grant dedicated monitoring virtual machines the rights to access the main memory of other virtual machines in order to run introspection operations. This allows customers to create monitoring virtual machines that have access to perform VMI-based operations on their production virtual machines. With our prototype implementation, we show that our approach does not introduce performance drawbacks and gives cloud customers full control to do introspection on their virtual machines. We also show that the impact of successful attacks to the monitoring framework is reduced.**

Keywords: Cloud Forensics, Virtual Machine Introspection, Mandatory Access Control, Xen

I. Introduction

Cloud computing has become an important paradigm in computer science in the last decade. The flexibility and scalability of this approach convinced many companies to move their resources to cloud environments. However, by moving data and computing to public cloud environments, customers lose the ability to perform security-relevant operations that are useful for intrusion detection systems such as virtual machine introspection (VMI) [11] where security software monitors and analyzes the high-level system state using the raw data in main memory of a virtual machine (VM) (the "guest system") [15]. Compared to in-guest security agents, the VMI approach has the advantage that the monitoring software is isolated from the guest and does not depend on functionality of the guest operating system, which can be tampered by an adversary (e.g., with a rootkit) and might not reliably return correct information. Many practical examples show that VMI is a very useful approach for security applications such as digital forensics [22], malware analysis [16] and intrusion detection [17].

There have been various attempts that bring VMI to Infrastructure-as-a-Service (IaaS) cloud environments and run forensic operations on virtual machines. CloudVMI [3] is a remote procedure call (RPC) wrapper for the introspection library LibVMI [1]. LiveCloudInspector [27] extends the API of the cloud management system with VMI-based forensics functions, such as taking snapshots of main memory and the execution of volatility scripts for analysing VMs. Both approaches have significant disadvantages in terms of performance, because the instruction (e.g., read memory) and the result of the operation must be marshalled and unmarshalled and transmitted, e.g., via network.

Another problem of these approaches is that the core part of the VMI functionality is running in the most privileged domain (such as the Dom0 VM on a Xen hypervisor). VMI-based systems heavily depend on the interpretation of data extracted from a guest system [4]. If an attacker is able to exploit flaws in that VMI functionality, he might be able to gain access to the privileged domain which has full access to all VMs running on the same physical node [6], [8].

There are several significant problems that make it impractical to use VMI in current cloud environments. First, in most (if not all) real cloud infrastructures, there is a **lack of access** for the cloud customer to VMI functionality. In addition, there is no established **billing model** for using VMI services. The few available research prototype solutions for the access problem are insufficient due to **lack of functionality** and **lack of performance**. And finally, **security risks** are an additional show-stopper for enabling VMI in production cloud infrastructures.

We present the CloudPhylactor[1] architecture, which is an approach to solve these problems by leveraging mandatory access control (MAC). In particular, we show how the implementation of the Flux Advanced Security Kernel (Flask) architecture of the Xen hypervisor can be used to grant a dedicated monitoring VM the rights to run VMI on other guest VMs. This enables cloud customers to use the Xen API for VMI on their VMs. Additionally, the access of these monitoring VMs can be restricted to a single guest VM or a subset of all VMs. Thus, even if an attacker is able to gain access to the monitoring VM, he will not have full control over the physical cloud node. Thereby, we increase the security of VMI-based monitoring approaches. In contrast to other cloud forensic solutions like CloudVMI [3] and LiveCloudInspector [27], CloudPhylactor does not introduce a significant overhead to the monitoring process, as it can

[1]The name CloudPhylactor is a combination of Cloud and the Greek word $\varphi\upsilon\lambda\alpha\kappa$ (phulak), which means watch or guard.

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.160

access the Xen interface directly and is therefore not restricted to use another more limited API. Moreover, this approach does not require fundamental changes to the cloud management software.

The remainder of this paper is structured as follows: Section II provides required background knowledge about VMI, MAC and Xen. In Section III, we discuss the threat model for VMI-based intrusion detection systems in cloud environments. In Section IV, the concept of the CloudPhylactor architecture is presented and the implementation details are described in Section V. Section VI discusses the performance and security of CloudPhylactor. Section VII gives an overview of related work. Section VIII summarizes the paper.

II. Background

In this section we provide background information in the fields of virtual machine introspection, mandatory access control and an overview of the architecture of the Xen hypervisor.

A. Virtual Machine Introspection

VMI is a set of techniques whose objective is to extract high-level information about VM status (e.g., the process list from an operating system) from low-level data sources (e.g., raw main memory). It is an approach of inspecting virtual machines from outside of the guest OS in order to analyze programs running inside of it. The gap between low-level data and the corresponding high-level state is described as the *semantic gap*. It can be further divided in the weak and the strong semantic gap. The first is considered a solved engineering problem as it is the process of interpreting data based on known structures. The second one describes the problems that arise when an attacker tries to place crafted data in memory to mislead the analysis tool [15].

Currently, the most applied tools for memory analysis are Volatility and Rekall [2], [26]. They provide many plug-ins that make it easy to perform a static analysis on main memory. LibVMI is a library that provides a unified interface to access the memory of VMs of KVM and Xen [1]. For performing dynamic or continuous monitoring, the tool DRAKVUF [18] can be used. It provides mechanisms to trace processes in a VM, e.g., by monitoring system calls or the allocation of memory.

B. Mandatory Access Control

In contrast to discretionary access control (DAC), the security of MAC is not defined by the owner of an object but by the "sensitivity [...] of the information contained in the objects" [9]. The decision whether access is granted or not is based on rules (policies) and on the security label of an object. Thus, an authority has to assign every subject and object one label based on meta knowledge of the system such as the confidentiality of the object.

A popular implementation of a MAC system is security enhanced Linux (SELinux) [20]. It is an implementation of the Flask security architecture [25] in the Linux kernel, which uses the Linux security modules (LSM) framework of the Linux kernel. It is a combination of *type enforcement (TE), role-based access control (RBAC)*, and optionally *multi-level security (MLS)* and *multi-category security (MCS)*.

A precondition for MAC and SELinux is that each subject (e.g., processes) and object (e.g., files, sockets, devices, IPC) must be labeled with a *security context* which consists of a user, role, type and optionally the security level/category. A SELinux security context has the form:

```
user:role:type:[level:category]
```

The default policy is to deny access. To give a subject (source) access to an object (target), a rule must be defined. Each policy rule has the form:

$$\texttt{allow}\ T_s\ T_t\ :\ C_a\ P_a$$

T_s specifies the set of source types, T_t the set of target types, C_a a set of classes of the control and P_a the set of permissions [12]. A class can be, for example, a file or socket and the corresponding permission could be read or write.

The policies of SELinux are static and must be compiled before they can be loaded to a running system. Thus, it is not possible to add new rules or types at runtime without reloading all rules. Policy rules can prohibit the reloading of rules, in which case the application of new rules requires rebooting the system.

C. The Xen Hypervisor

Xen is a Type 1 or bare metal hypervisor [23] and one of the most important virtual machine monitors (VMMs) as it is used in many IaaS cloud data centers such as the clouds of Amazon and Rackspace.

At boot, Xen starts the domain 0 (Dom0) which is a privileged domain that manages other unprivileged domains (DomU). Dom0 has access to the physical devices and provides virtual devices to DomUs. Therefore, the guest has to implement a front-end driver and the Dom0 the back-end driver. In general, Dom0 is a paravirtualized fully fledged Linux system and in cloud infrastructures, the node management software is installed in it.

Xen uses different kind of channels for the communication between the hypervisor and domains and between domains. The back- and front-end drivers exchange messages with each other via intra-node or inter-domain communication [19], which is established with shared memory regions. Therefore, each domain can grant other domains access to their memory pages. The information about shared pages is stored in *grant tables* for each domain.

Xen, since version 4.3, comes with its own implementation of the Flask architecture, which uses the Xen security modules (XSM) interface [7]. This implementation controls the access and communication of Xen domains, the Xen hypervisor and resources such as devices or memory, but not of applications and files. The XSM Flask implementation uses the same language for defining the policies as SELinux, however a different set of rules, users, roles and types is used. While previous versions of Xen did not allow executing hypercalls

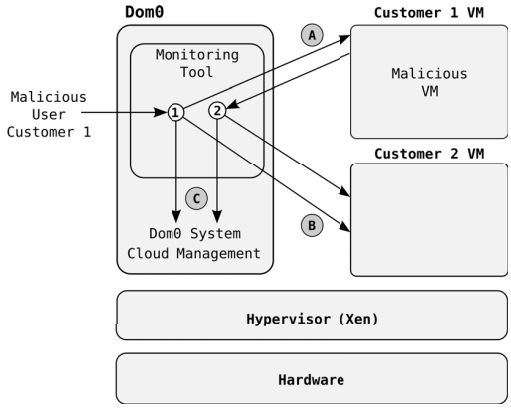

Fig. 1. Threat model of VMI in cloud environments

in unprivileged domains, the implementation of MAC makes it now possible to restrict the access between domains in a more fine-grained way, including the possibility to grant selected VMs more permissions than regular DomUs.

III. Threat Model and Assumptions

This section describes the two most important attack vectors to VMI-based monitoring tools in cloud environments. And later a third attack vector for cloud computing – the malicious cloud admin – is discussed. The threat model is depicted in Figure 1 where (A) is the legitimate access of a (malicious) user on his (malicious) VM. All other cases are not legitimate.

The first attack targets the (user) interface of the monitoring tool (1). Possible attacks are subverting the access control of the monitoring tool or exploiting software flaws in the implementation, e.g., buffer overflows. If an attacker has successfully attacked the monitoring tool, he can use its permissions to access other VMs. As the analysis tool is usually installed in the privileged domain an attacker gets access to all VMs that run on the same cloud node [3], [27]. This attack requires that an attacker has access to the interface of the monitoring tool.

The second attack targets the interpretation of main memory of VMs (2). Therefore, we assume that an attacker has full control over a VM and can modify all entries in memory. VMI-based systems heavily depend on the interpretation of data extracted from a guest system. DKSM demonstrates that manipulating a VMI-based system by changing in-guest data structures is easily feasible, e.g., for place misleading or wrong information to fool the analysis tool or to hide traces [4].

The problem of incorrect interpretation of the state of a VM that has been manipulated by an attacker is known as the *strong semantic gap* and needs to be addressed by the monitoring tool [15]. This problem, which concerns the validity of the results obtained by the monitoring tool, is outside the scope of this paper. Our focus is on preventing any malicious impact outside of the monitoring tool.

While this example is limited to the manipulation of analysis results, it is highly likely that VMI-based monitoring systems contain vulnerabilities that can be used to compromise the monitoring tool. Similar attacks already exist for Tripwire [8] and Snort [6]. They show that an attacker is able to interact with a target system in a way such that the collected monitoring data exploits vulnerabilities in an intrusion detection system. In the case of VMI, a flaw in the interpretation of the monitoring framework can be used by an attacker to get access to the system where the monitoring framework is running. If the monitoring tool is installed in Dom0 an attacker gets access to the system running in Dom0 (C) or to VMs of others users (B).

A more general threat to cloud computing is a malicious cloud administrator. This threat is out of the scope of this paper. Nevertheless, we will discuss in Section VI to what extent our proposed approach with a MAC-based system can contribute to solving this malicious insider problem.

IV. CloudPhylactor System Design

Our proposed architecture – CloudPhylactor – allows cloud customers to run VMI-based operations on their production VMs. Furthermore, it minimizes the risk of privilege escalation, when an attacker exploits flaws in the monitoring software and gains control over the VM that runs the VMI software. For this purpose, we move the VMI-based operations into a dedicated VM with limited access to other domains (see Figure 2). In the following, we call a VM with normal permissions *production virtual machine (PVM)* and a VM with permissions to run VMI *monitoring virtual machine (MVM)*. Additionally, we use the term *user* in the context of a cloud customer and not for different entities such as administrators and forensic investigators. Instead, we are discussing the permissions of VMs and not entities that are using them. The mapping of entity roles to policy users is out of the scope of this paper.

The concept that we are describing is independent of a specific hypervisor. However, only Xen supports MAC or at least a fine-grained access control that supports to define that a VM has permissions to access another VM's main memory. Thus, we use the terminology of Xen in the architecture description.

A. Monitoring Virtual Machine

A MVM must have at least the permissions to execute these operations on another domain for VMI:

- *read access on main memory* to extract low level information, e.g., to extract the process list of the operating system.
- *read access on CPU registers* to extract the current CPU state, e.g., to determine the process which is currently running on a CPU. For example, the CR3 register contains a pointer to the memory mappings of the process that is active.

These permissions are sufficient for extracting information from a virtual machine (e.g., the process list) for static analysis or when the contents in main memory should not be modified,

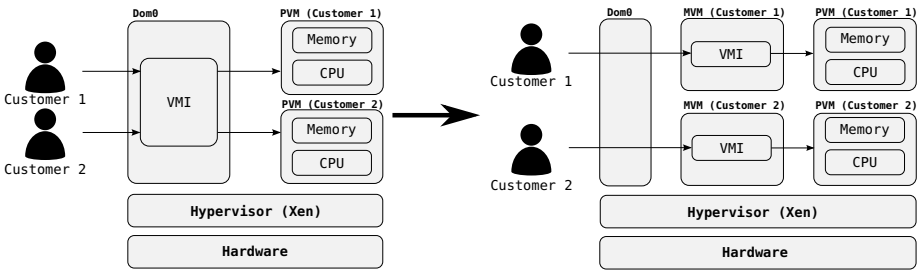

Fig. 2. The left side represents the most used architecture: all VMI-based operations are executed in the privileged domain. The right side represents the CloudPhylactor architecture where a domain is monitored by a dedicated monitoring VM. For the right side the user needs to have access only to his MVM and not to a program in Dom0.

e.g., for forensic evidence collection. To trace the execution of a VM writing permissions are required:

- *write access to main memory* to modify program code, e.g., to insert software breakpoints which are necessary to trace the execution of a process or to inject code. A software breakpoint is an instruction that causes the CPU to throw an interrupt.
- *write access to CPU registers* to change CPU state, e.g., to change return values.
- *access the Xen event channel;* it allows monitoring VM events such as interrupts of a VM which can be caused by software breakpoints.

A MVM should contain only necessary tools to minimize the attack surface [24]. The operating system of a MVM can be either a fully-fledged Linux system with several forensic tools or a minimal domain, where the kernel is the forensic tool. The last approach helps to minimize the amount of additional resources and reduces the attack surface to a minimum.

B. Mandatory Access Control

The goal of CloudPhylactor is to grant MVMs the permissions to run VMI-based operations on PVMs by using the MAC of Xen. Therefore, we introduce a new Flask type for MVMs. The Flask type is part of the label which is assigned to a VM when it is launched. It cannot be changed during runtime. This label also includes the name of a Flask user. We use it to constrain the access of MVMs so that they can access only PVMs of the same user but not of others. The enforcement of the policies is performed at runtime by the Xen hypervisor. As Flask users are part of the policy which is static, all users must be known before they are compiled. Thus, we create dummy users in the Flask database and map cloud customers to them. The mapping process is performed by the cloud management.

C. Cloud management

The cloud management is the interface between the user and the cloud infrastructure, e.g., Xen. Thus, it has to provide a user interface which allows setting the type (PVM or MVM) of a VM and the group of PVMs that can be monitored by a MVM. When a VM is launched, the cloud management has to compute the security label of it based on the user input. Additionally, it has to manage the mapping from cloud users to dummy flask users.

V. Implementation

For our implementation of the CloudPhylactor architecture, we use OpenNebula 4.12.3 as a cloud management software and Xen 4.5 as a hypervisor. To use MAC for our architecture, we define a new policy type for MVMs and its permissions on PVMs. Then we assign domains to users so that a MVM is able to access only domains of the owner but no others. Additionally, we generate Flask users and map them to cloud customers. We will discuss these steps in detail in the following paragraphs.

A. Policies

To grant a MVM access to the main memory of other VM, we introduce a new domain type – `domU_monitoring_t` in the Xen Flask policies and derive it from the permissions of a regular DomU. Furthermore, we define rules that grant a MVM the permissions to execute VMI-based operations on domains of the type `domU_t` (see Section IV-A).

B. User management

When a domain is started in the context of a `domU_monitoring_t`, it is able to perform all granted operations on every domain running in the context of `domU_t` even if the VM belongs to another customer. Usually such global access is unwanted, and instead access should be restritected on a per-customer basis. Therefore we restrict VMI permissions to the domains that belong to the same user or to only a subset of his domains if a more fine-grained restriction is required.

As the policies are static and cannot be modified, all users must be known before the rules are compiled. Otherwise, rules must be adapted, compiled and loaded whenever a new user registers. As the user set of cloud providers is changing permanently and reloading of rules can introduce security and performance drawbacks, this approach is not feasible for production environments.

To prevent the generation of Flask users at run-time, we create a set of dummy users and map cloud users to them. The mapping can be *cloud wide* or *cloud node local*. A cloud wide mapping maps each customer to the same Flask user on

all nodes. This requires that all cloud nodes must have at least as many Flask users as real cloud customers. This approach does not scale for very big installations, as every cloud node has to provide Flask users for all cloud users. Cloud node local mapping solves this problem. As the mapping is local, a user might be mapped to a different Flask user on every cloud node or might not be mapped at all when he has no VM running on this node. In both cases, the cloud management must handle the mappings.

Additionally, not all monitoring domains of one user should be able to monitor all PVMs of that user. In fact, if VMs contain data of different levels of clearance and an attacker is able to subvert their monitoring domain, he can also gain access to data with a higher level of clearance. To solve this problem, VMs can be assigned to sub-users where access is only granted in between sub-users. For example, one cloud customer has the user ID `1`. Then the sub-users would be `cloud_customer_1_{0..n}`. By choosing a sub-user ID, the cloud customer is able to define the group of VMs that can be monitored by a MVM as it is only able to monitor other VMs with the same sub-user ID.

Another approach is the usage of MLS/MCS policies. This concept assigns each unit (i.e., domain) a sensitivity level and a category. It is based on the model of Bell and LaPadula [13], [5]. However, the MLS and MCS support of Xen is incomplete.

C. Cloud Management

We extend OpenNebula so that a cloud customer is able to create monitoring machines by adding two flags to the template of a VM. One flag defines whether a VM is a MVM and the other one defines the sub-user ID. Both flags are translated in the Xen deployment configuration to the corresponding security label. Furthermore, we implement a mapping mechanism from OpenNebula to users in the Flask policies. Therefore, we use a one-to-one mapping that translates the OpenNebula user ID to a Flask user with the same ID. The computed security label of a VM has the form:

```
customer_{userid}_{class}:vm_r:{type}
```

The variable `userid` is defined by the cloud management and derived from the ID of an OpenNebula user. The `class` variable and the `type` of a VM is defined by the customer in the template variables of a VM. Moreover, the cloud customer can use the user interface to choose on which cloud node his VM shall be launched in order to place the MVM on the same cloud node where the PVM is running.

D. Monitoring VM

Another part of the CloudPhylactor architecture is the MVM. In our implementation it is a fully fledged Linux which contains all necessary libraries to communicate with the Xen hypervisor. Additionally, we have installed tools such as LibVMI [1] and Volatility to run standard forensic operations [26]. Finally, the MVM must be paravirtualized. Otherwise, some hypercalls required for VMI are not available to the guest system.

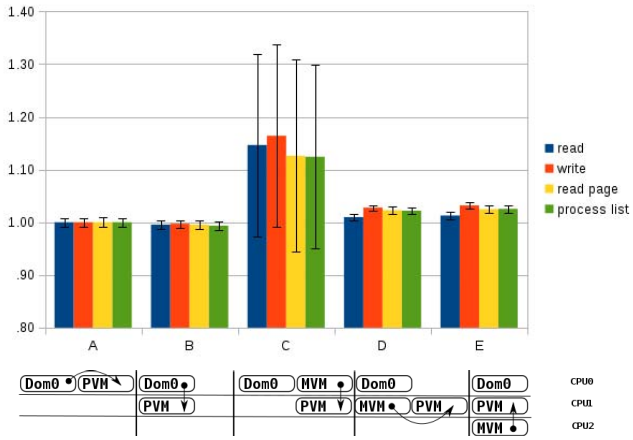

Fig. 3. Relation between the time that is required for the VMI access methods of the testcases B to E compared to testcase A

VI. Evaluation and Discussion

In this section, we measure the performance of the CloudPhylactor architecture and discuss its security. All described tests are executed on a server with an Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz CPU with six cores and 32 GB RAM.

A. Performance

To determine the performance difference between the access from Dom0 and from a MVM, we run four tests that mimic the behavior of often used real world VMI operations. These are:

1) A four byte read operation which is often used to retrieve integer values or pointers
2) A four byte write operation which can be used to insert software breakpoints
3) A read operation that requests a full page (4096 Bytes), e.g., to take a snapshot of process memory
4) Extraction of PVM's process list

These are real-life use case scenarios which can be used to detect suspicious processes or for doing forensic operations on main memory. Even if more complex VMI-based monitoring tools are used, they rely on the same operations as the extraction of the process list requires, i.e., the read operation to extract data structures. Using more complex VMI operations would partially conceal and clutter with noise the observed overhead values. The extraction of the process list is not trivial and uses most common LibVMI features for parsing and traversing kernel data structures and we consider it representative for a VMI tool that analyses guest kernel data structures.

We measure the performance of the VMI operations for the four test cases in several different configurations. The VMI operations are either executed (as done traditionally) in Dom0, or (using our CloudPhylactor architecture) in MVM. We also use different mappings of virtual machines to physical CPUs in order to find out if such mapping has an impact on performance, e.g., due to some cache effects. In total, the

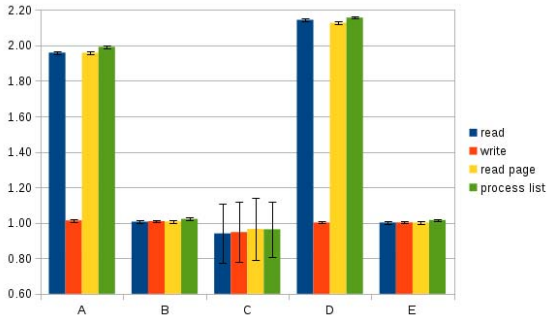

Fig. 4. Relative access time of the VMI access methods when the PVM is running a Benchmark compared to testcase of Figure 3

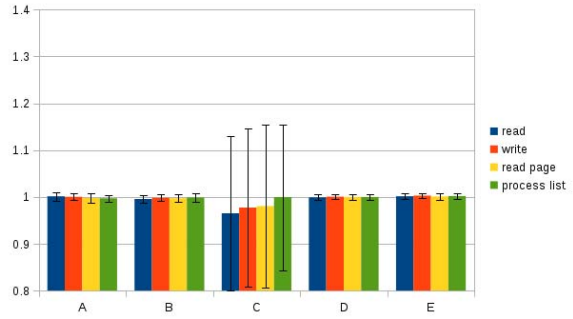

Fig. 5. Relative VMI access time with 10,000 flask users compared to same measurement with 1,000 users

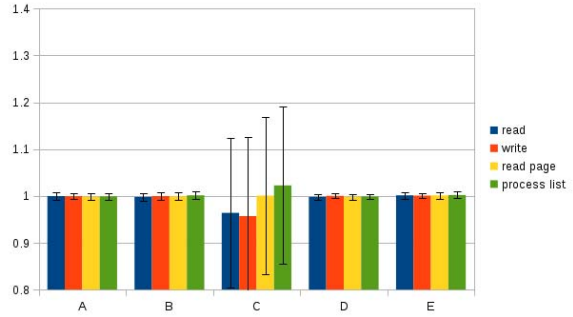

Fig. 6. Relative time of VMI access time of the measurements A to E with 50,000 flask users compared to same measurement with 1,000 users

following five configurations are used, and VMI operations are executed from:

A **Dom0:** PVM is pinned to the same CPU as Dom0
B **Dom0:** PVM is pinned to a different CPU then Dom0
C **MVM:** Dom0 and MVM are pinned to the same CPU; PVM is pinned to another CPU
D **MVM:** PVM and MVM are pinned to the same CPU; Dom0 is on another CPU
E **MVM:** Every VM is pinned to another CPU

To prevent the caching from LibVMI we disable this feature for all measurements. During this measurement the PVM is mostly in idle state. For the access time, we measure the time that is required for 10,000 operations and take the average and standard deviation of 100 runs. Figure 3 shows the results of this measurement. We use configuration A (VMI on Dom0, on same CPU as PVM) as baseline, and for all other configuration, we plot the relative execution time of the VMI operations (smaller values are better). We can conclude that even in the worst case the VMI performance loss is less than 3%. The overhead of about 17% percent for the testcase C is caused by the fact that Dom0 and MVM are pinned to the same CPU. Thus, the VMI access is delayed by the execution of Dom0.

The low access latency is an advantage compared to the CloudVMI [3] solution as their approach uses remote procedure calls. Thus, depending on where the measurements are executed – on the same or on a remote cloud node – they experience a high overhead up to two hundred percent. This causes a negative performance impact when events are used, e.g., to monitor software breakpoints. In that case, the execution of the PVM is paused until the event is processed by the monitoring tool. Thus, when the execution is delayed by the latency of the network, the performance of the PVM decreases.

To evaluate the impact which is caused by the placement of the VMs when the VM is polluting the CPU cache, we run the same measurements while the PVM is running a benchmark. Therefore, we run the Dhrystone and pipe-based context switching benchmark of the byte-unix benchmark suite[2] in a loop in the PVM. The results of the relative access times are depicted in Figure 4, where we use the measurements of Figure 3 as baseline. When the VM running the VMI operations shares the CPU with the PVM, we have a significant impact for the read, readpage and process list operation but not for the write access. This is caused by the measurement method: we pause the PVM for the write operation in order to avoid inconsistency of the VM state as we write the value that has been read before. Thus, the PVM is not scheduled during the VMI operation and does not affect the time for the VMI access. The other operations do not pause the PVM. Thus, the VMI operation can be interrupted by the PVM that is doing the CPU intense benchmark.

B. Policy

Another aspect that might influence the performance of VMI-based operations is the size of the policy data base. The Xen hypervisor checks for each access if it is allowed or not. Therefore, it has to look up each decision in the database. To check if the lookup takes longer for a bigger database we run the same measurements as in Section VI-A but with a bigger policy database, i.e., 10,000 and 50,000 users instead of 1000 users. Figure 5 and 6 shows the results for these measurements. Based on these measurements, we conclude that there is no significant difference at runtime between a big policy database and a small one.

[2]Byte-unixbench - https://github.com/kdlucas/byte-unixbench Accessed on 10 December 2015

Users	1,000	10,000	50,000
Compile (s)	0.20 (0.03)	2.90 (0.14)	50.93 (2.33)
Load (s)	0.0035 (0.01)	0.30 (0.02)	5.63 (0.20)
Size	96 K	860 K	4.2 M

TABLE I
AVERAGE TIME IN SECONDS AND STANDARD DEVIATION TO COMPILE AND LOAD POLICIES MEASURED IN RELATION TO THE AMOUNT OF USERS.

Additionally, we measure the size of the database and the time that is required to compile and load the policy database. The results in Table I show that the size correlates linear with the amount of users but the time does not. These measurements show that both global and local user mapping is feasible. However, the policy database should not become too big, otherwise it is not possible to load it anymore to main memory. In that case, it is necessary to switch to a local mapping to keep the policy database small on each cloud node (see Section V-B).

C. Resources

Compared to LiveCloudInspector our approach requires – in the worst case – a separate MVM for each PVM instead of only a monitoring process. Of course, a MVM can monitor more than one PVM, however it is not recommended for security reasons. Thus, additional resources such as main memory, CPU time and storage are necessary for each MVM. The amount depends on the configuration and the memory that is required by the processes that are running in the VM. However, techniques such as memory deduplication can be used to decrease the amount of required physical memory [21]. Additionally, MVMs can be stripped down to a minimal system, e.g., use only a very minimalistic kernel. Moreover, with VMI on Dom0 it is impossible to attribute CPU and memory consumed by monitoring to a customer. With MVM, the cloud customer can rent a MVM of appropriate size (small for lightweight VMI monitoring, larger for more complex processing of VMI data), and the billing infrastructure of the cloud can simply be reused without adding any additional complexity.

D. Security

1) Attack impact: In Section III, we describe the risks that come along with the installation of an analysis tool in a privileged VM. This includes the access to another VM or the execution of commands in the privileged domain. By moving the analysis tool to a dedicated MVM we protect the cloud management infrastructure and other cloud customers. In the worst case, an attacker is only able to instrument the analysis tool in a MVM and execute commands in this domain or to access the PVM of the same customer. But he is not able to attack other parts of the cloud infrastructure.

2) Malicious insider: MAC can be used to protect customer's VMs from malicious administrators as well. For example, the cloud management interface can be installed in a dedicated management VM that is only able to start and stop VMs and without permission to access the main memory of a VM. However, if an administrator is able to start MVMs with the security label of another customer, he can use that mechanism to gain access the main memory of a VM. When the cloud environment does not provide MVMs, MAC is a feasible way to protect cloud customers from the access of malicious administrators to the memory of a VM because the access of the privileged domain of the cloud management can be restricted so that an administrator is not able to use it for malicious VMI.

VII. RELATED WORK

CloudVMI [3] provides VMI to customers through the functionalities of LibVMI by using remote procedure calls (RPCs). To monitor one or more of his VMs, a client can use the cloud management to start a monitoring VM hosting the VMI-based application and a virtualization of LibVMI interface called the vlibVMI client library. The cloud management maintains database entries about users and the policies that map each user to its VMs. Unlike our design of MVM, their approach suggests that for each monitored VM, the associated service module hosted by the privileged monitoring VM (Xen) receives the RPC calls from the client. While additional latency is introduced by the use of RPC and (network) communication, monitoring tools that use CloudPhylactor do not have this latency as they can directly access the Xen API.

The LiveCloudInspector [27] architecture provides and interface for customers for VMI-based remote host forensics and network analysis. The paper presents a method for live investigation on running systems, while respecting the strict necessity of separating the multiple tenants that share the same physical hardware support. Therefore, a dedicated forensic platform is installed in Dom0 of a cloud node. In contrast to CloudPhylactor, cloud customers are only able to run predefined operations on their VMs. This includes for example the execution of Volatility script for the creation of memory snapshots. However, customers do not have full access to their VMs with LibVMI or the Xen interface.

Harrison et al. [14] describe the concept of forensic virtual machines (FVMs) for the first time. They define FVM as a lightweight VM configured to inspect VMs (one at a time) and find a defined malware symptom. FVM's special privileges are granted by the VMM. Using the XenAcces library, it is also possible to create VMs that can monitor other VMs or access their physical resources via Dom0's management interface.

Adrian et al. built a prototype that uses FVMs [24]. Their prototype is designed for the Xen virtualization platform. Communicating via secure multi-cast channels, the FVMs also provide dynamic defence through the detection of symptoms of malicious behavior. The access control of Xen has been modified to allow the FVMs inspect other domains. But it highlights the need for suitable access control mechanisms that determine the inter VM access.

In the design of FROST, the digital forensic tools for OpenStack platform [10], data is collected at host OS level at the cloud provider, then made available in OpenStack management plane, accessible through API and website. Their

method allows the user to retrieve an image of virtual disks and OpenStack Firewall logs for his VMs. Each user's API requests are independently stored in a tree structure, with layer corresponding to machine instance time-stamping. To provide information integrity, the hash values of collected forensic data are periodically calculated.

VIII. Conclusion

In this paper, we presented the CloudPhylactor architecture. It gives cloud customers the possibility to run VMI-based operations on their VMs in cloud environments. Compared to other approaches, our architecture is easy to implement and does not introduce latency to VMI-based monitoring tools. Thus, it is feasible for large scale public IaaS-based cloud systems. Another contribution of this paper is that CloudPhylactor enhances the security of existing VMI-based tools. The common VMI approach is to run tools in a most privileged environment (Dom0 for Xen-based systems). As stated in Section VI, we try to minimize the attack vector and the impact of attacks against the monitoring entity by restricting the access of the MVM. In our opinion, the approach of the CloudPhylactor architecture is currently the most secure way to use VMI in virtual environments. Furthermore, while we can configure our system to permit write access from MVM to a PVM, we can also irrevocably restrict a MVM to read-only access, if that is preferred due to security considerations (see IV-A).

Another aspect of the CloudPhylactor architecture is that cloud providers are now able to provide *VMI-as-a-service*. In contrast to other solutions like LiveCloudInspector, billing is easier to accomplish for VMs than for processes in Dom0 as no additional implementation is required because the billing infrastructure of regular VMs can be applied.

Acknowledgment

This work has been supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association and by the German Federal Ministry of Education and Research (BMBF) in the project DINGFEST-EFoVirt.

References

[1] LibVMI. http://libvmi.com/. Accessed on 17 November 2015.

[2] Rekall - Memory Forensics Analysis framework. http://www.rekall-forensic.com. Accessed on 25 November 2015.

[3] H.-w. Baek, A. Srivastava, and J. V. D. Merwe. CloudVMI: Virtual Machine Introspection As a Cloud Service. In *Proceedings of the 2014 IEEE Int. Conf. on Cloud Engineering*, IC2E '14, pages 153–158, Washington, DC, USA, 2014. IEEE Computer Society.

[4] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: subverting virtual machine introspection for fun and profit. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 82–91, Washington, DC, USA, 2010. IEEE Computer Society.

[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[6] CERT. Multiple vulnerabilities in snort preprocessors. https://www.cert.org/historical/advisories/CA-2003-13.cfm, Apr 2003. Accessed on 25 November 2015.

[7] G. Coker. Xen Security Modules (XSM). http://mail.xen.org/files/summit_3/coker-xsm-summit-090706.pdf, March 24 2015. Accessed on November 17 2015.

[8] Common Vulnerabilities and Exposures. CVE-2004-0536. Available from MITRE, CVE-ID CVE-2004-0536., 2004. Accessed on 11 April 2016.

[9] Computer Security Center (U.S.). *Computer Security Requirements: Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments*. CSC-STD. DOD Computer Security Center, 1985.

[10] J. Dykstra and A. T. Sherman. Design and implementation of FROST: Digital forensic tools for the OpenStack cloud computing platform. *Digit. Investig.*, 10:87–95, 2013.

[11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[12] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying Information Flow Goals in Security-enhanced Linux. *J. Comput. Secur.*, 13(1):115–134, Jan. 2005.

[13] C. Hanson. SELinux and MLS: Putting the pieces together. In *Proceedings of the 2nd Annual SELinux Symposium*, 2006.

[14] K. Harrison, B. Bordbar, S. Ali, C. Dalton, and A. Norman. A framework for detecting malware in cloud by identifying symptoms. In *IEEE 16th Int. Enterprise Distributed Object Computing Conference (EDOC)*, pages 164–172, Sept 2012.

[15] B. Jain, M. Baig, D. Zhang, D. Porter, and R. Sion. SoK: Introspections on Trust and the Semantic Gap. In *IEEE Symposium on Security and Privacy (SP), 2014*, pages 605–620, May 2014.

[16] X. Jiang and X. Wang. "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 198–218. Springer Berlin Heidelberg, 2007.

[17] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[18] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proc. of the 30th Annual Computer Security Applications Conference*, 2014.

[19] N. Li and Y. Li. A Study of Inter-domain Communication Mechanisms on Xen-based Hosting Platforms. http://www.cs.ucsb.edu/~nanli/projects/cs270.pdf, 2009. Accessed on 11 April 2016.

[20] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[21] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[22] K. Nance, M. Bishop, and B. Hay. Investigating the Implications of Virtual Machine Introspection for Digital Forensics. In *Proceedings of the Int. Conf. on Availability, Reliability and Security, 2009. ARES '09*, pages 1024–1029, March 2009.

[23] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[24] A. L. Shaw, B. Bordbar, J. Saxon, K. Harrison, and C. I. Dalton. Forensic virtual machines: Dynamic defence in the cloud via introspection. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 303–310, Washington, DC, USA, 2014. IEEE Computer Society.

[25] R. Spencer, S. C. Corporation, S. Smalley, P. Loscocco, N. S. Agency, and M. H. D. Andersen. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of The Eighth USENIX Security Symposium*, pages 123–139, 1999.

[26] Volatility Foundation. Volatility - open source memory forensics. http://www.volatilityfoundation.org/ Accessed: 21 April 2016.

[27] J. Zach and H. P. Reiser. LiveCloudInspector: towards integrated IaaS forensics in the cloud. In *Proc. of the 15th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*, 2015.