Towards Hypervisor Support for Enhancing the Performance of Virtual Machine Introspection

Benjamin Taubmann(✉) and Hans P. Reiser

University of Passau, Passau, Germany
{bt,hr}@sec.uni-passau.de

Abstract. Virtual machine introspection (VMI) is the process of external monitoring of virtual machines. Previous work has demonstrated that VMI can contribute to the security of cloud environments and distributed systems, as it enables, for example, stealthy intrusion detection. One of the biggest challenges for applying VMI in production environments is the performance overhead that certain tracing operations impose on the monitored virtual machine. In this paper, we show how this performance overhead can be significantly minimized by incorporating minor extensions for VMI operations into the hypervisor. In a proof-of-concept implementation, we demonstrate that the pre-processing of VMI events in the Xen hypervisor reduces the monitoring overhead for the use case of VMI-based process-bound monitoring by a factor of 18.

1 Introduction

Virtual machine introspection (VMI) is the process of analyzing the state of a virtual machine from outside, i.e., the perspective of the hypervisor [5]. Based on the external view, VMI-based monitoring has certain properties that make it appealing for many application scenarios, including solutions that aim to enhance the security in cloud environments and distributed systems [6,8]. Hence, it is not surprising that the first application for VMI was an intrusion detection system [5]. Those properties are: isolation between the monitoring and the monitored system, an untampered view on the system state, and the stealthiness of monitoring.

In the following we use the term *production virtual machine* for the virtual machine that is monitored via VMI and is performing the normal operations. The term *monitoring virtual machine* is used for the virtual machine that analyzes the production virtual machine using VMI [18]. The focus of this paper is on the Xen architecture. Hence, the term monitoring virtual machine refers to either the Dom0 of Xen or any other virtual machine with the permissions to perform VMI-based operations. The VMI application that performs the analysis does not run directly in the hypervisor.

Published by Springer Nature Switzerland AG 2020
A. Remke and V. Schiavoni (Eds.): DAIS 2020, LNCS 12135, pp. 41–54, 2020.
https://doi.org/10.1007/978-3-030-50323-9_3

There are two forms of virtual machine introspection: *asynchronous and synchronous*. The main difference between these operation modes is how the analysis is started [8]. We speak of asynchronous virtual machine introspection if the analysis is started by external events, such as timers that periodically retrieve the system state. This is useful to regularly observe the system state and retrieve information that is in memory for a longer time frame, such as the list of running processes. Synchronous virtual machine introspection is triggered by sensitive operations in the control flow of the monitored system, e.g., software breakpoints. This can be required to retrieve ephemeral information that resides in main memory only for a short period, such as the parameters of function calls.

In this paper we use the term *performance impact* to quantify the overhead induced on the execution in the production virtual machine due to VMI-based operations. The overhead is mainly caused by the fact that the production virtual machine is paused for the analysis, which has three different reasons. The first one concerns the case when the production virtual machine and monitoring virtual machine share the same physical CPU core. In this case, the production virtual machine gets paused when the monitoring system is running. This overhead factor can be reduced by using an additional CPU core or by minimizing the amount of time required for the analysis.

The second reason is that the production virtual machine must be paused to prevent that the analyzed data structure (e.g., the process list) is modified during the analysis. If the analysis were performed on a running system, memory contents changing concurrently to the analysis could result in incorrect information. In order to obtain reliable information, it is, therefore, necessary to perform the analysis in a non-modifying state in order to retrieve a consistent view of the system state. A common approach to address this problem is to take a snapshot of the contents in main memory and then run the analysis on it. However, taking a snapshot of the main memory may also require to pause the production virtual machine. Klemperer et al. [9] addressed this problem by implementing a snapshot mechanism that uses a copy-on-write strategy.

The third reason is that the production virtual machine is paused due to synchronous VMI-based mechanisms, e.g., when a breakpoint is invoked and the control flow of the production virtual machine traps to the monitoring virtual machine. The impact on the performance of the production virtual machine, in this case, depends on how often the monitored virtual machine is interrupted and the time how long the monitored system is paused for the analysis.

While the first two causes of performance overhead have adequate solutions, the third reason is an open problem that severely limits the applicability of synchronous VMI-based monitoring in practical use cases. In this paper, we focus on how to minimize the performance overhead in this third case and discuss mechanisms to enable efficient VMI-based monitoring. In detail, we discuss an approach that reduces the overhead of process-bound system call monitoring. Minimizing the overhead of synchronous VMI-based monitoring is important to implement VMI-based security solutions in production environments that tolerate only a minimal performance impact.

The contributions of this paper are:

- a presentation of different approaches that aim to minimize the performance impact of synchronous virtual machine introspection;
- a proof-of-concept implementation that minimizes the performance impact of process-bound monitoring;
- the evaluation of the proof-of-concept implementation.

The outline of the paper is as follows: Sect. 2 introduces the most important technologies used by virtual machine introspection and the most common mechanisms of synchronous virtual machine introspection. Section 3 discusses approaches on how to minimize the overhead of synchronous VMI-based monitoring. In Sect. 4, we discuss a prototype implementation that aims at minimizing the monitoring overhead of process-bound system call tracing. In Sect. 5, we measure the performance gain of the prototype. Section 6 discusses related approaches and Sect. 7 concludes the paper.

2 Virtual Machine Introspection

This section describes the technologies that are used for virtual machine introspection and synchronous monitoring.

2.1 Hardware Requirements

Synchronous virtual machine introspection requires that the CPU is able to trap to the hypervisor in order to monitor the execution of a virtual machine. For example, the hardware virtualization instruction set of Intel processors (Intel VT-x) allows trapping to the hypervisor (VM-Exit) when certain sensitive instructions are invoked in a virtual machine [7,20]. The virtual-machine control structure (VMCS) defines for each virtual machine in which case it should trap to the hypervisor (VMX root mode). While the hypervisor handles the event, the virtual machine that invoked the sensitive operation is paused.

2.2 Hypervisor Support

The hypervisor has an important role for virtual machine introspection. For asynchronous introspection, the hypervisor must give the monitoring virtual machine the permissions to access the main memory of the production virtual machine. This can be done for example by mapping the memory pages from the production virtual machine into the address space of the monitoring virtual machine. For synchronous monitoring, the hypervisor needs to forward the information about traps to the monitoring virtual machine so that it can run analysis operations. The concepts presented in this paper are mostly hypervisor independent. However, the focus of this paper is on the design of the Xen architecture.

Xen [1] is a bare-metal hypervisor that supports the hardware virtualization instruction set of Intel. The relevant components for VMI-based operations using

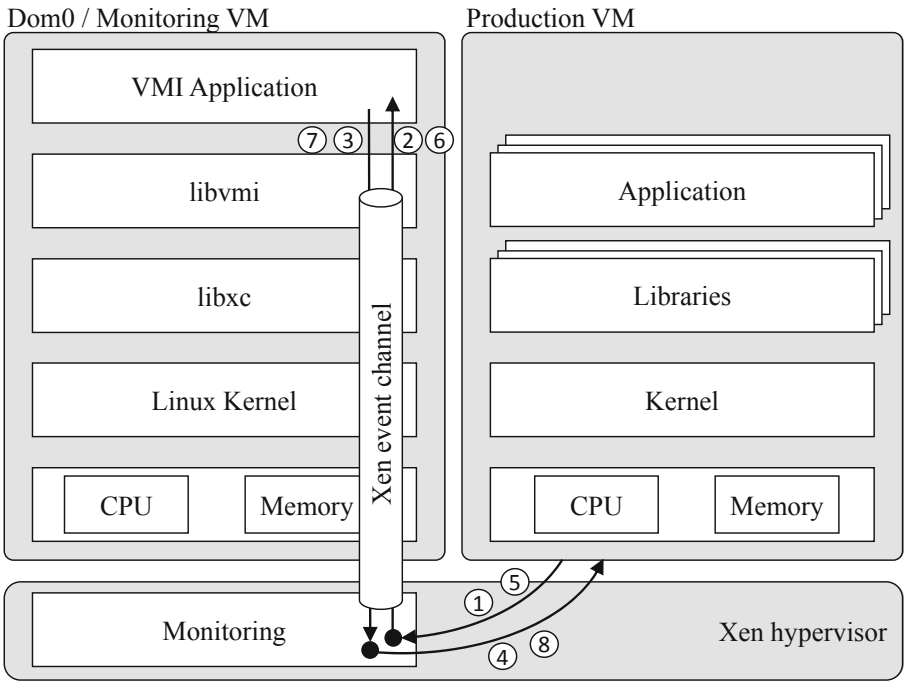

Fig. 1. VMI architecture using Xen and Libvmi. The arrows show the control flow of VM Exit/Entry-based context switches when a sensitive instruction in the production VM triggers a VMI event. The numbered items illustrate that synchronous tracing with breakpoints requires two iterations, the first to intercept the breakpoint and single-step the original instruction, the second to re-insert the breakpoint and resume the production VM.

Xen are depicted in Fig. 1. The Dom0 is the virtual machine with the most permissions and can access the main memory of all virtual machines running on this hypervisor. Thus, most VMI-based approaches implement the VMI application in Dom0. However, with the Xen security modules (XSM) it is possible to grant other virtual machines than the Dom0 the permissions required to perform VMI-based operations [17,18]. This has certain advantages and can be useful in cloud computing when it is necessary to restrict the VMI permissions of a monitoring virtual machine to the production virtual machine of a specific user.

Asynchronous VMI-based operations can be implemented with Xen by mapping the relevant memory pages of a virtual machine to the monitoring virtual machine, e.g., the Dom0. The functions that map memory pages are implemented in the libxencontrol (libxc) library and use Xen hypercalls to instruct the hypervisor. For synchronous VMI-based operations, sensitive operations in the production virtual machine cause a trap to the hypervisor. Then, the monitoring module of the hypervisor becomes active and sends the corresponding VMI event via the event channel to the VMI application in the Dom0/monitoring

virtual machine. While the event is processed in the monitoring virtual machine, the production virtual machine is paused.

2.3 VMI Applications

LibVMI [13,14] is the defacto standard library that helps to build VMI-based monitoring applications. LibVMI provides the API to perform VMI-based operations on Xen and KVM. This includes, for example, functions for reading and writing content from the main memory and functions for translating virtual and physical addresses. In addition, LibVMI supports synchronous monitoring with Xen. For this purpose, it provides primitives for listening on VMI-events generated by the Xen hypervisor.

2.4 Basic Synchronous VMI Methods

The main advantages of synchronous virtual machine introspection are that it can be used to monitor the control flow of virtual machines and to extract ephemeral information such as parameters of function calls. The mechanisms of synchronous monitoring utilize the concepts of hardware virtualization that allow trapping to the hypervisor under certain conditions, e.g., when sensitive instructions are invoked. In the following, we explain the most common VMI-based monitoring techniques: breakpoints, the monitoring of the write access to CPU control registers, and system call tracing.

Breakpoints. A common approach to monitoring the control flow using virtual machine introspection is to insert software breakpoints, by using the INT3 instructions that trap to the hypervisor when they are invoked. One approach to implement a software breakpoint mechanism for virtual machine introspection requires the following eight steps. First, the software breakpoint is inserted at the function that should be monitored, either by changing the content in memory or by manipulating the page tables[1]. Second, the production virtual machine invokes the function and the CPU traps to the hypervisor. Third, the hypervisor handles the trap caused by the INT3 instruction, creates a VMI monitoring event and sends it via the Xen event channel to the virtual machine handling the event. Fourth, the monitoring virtual machine gets active and the VMI application receives the new event. Fifth, the VMI application analysis the state of the production virtual machine and re-inserts the original instruction. The VMI application finishes the investigation and tells the hypervisor that the production virtual machine should perform a single-step and execute the original instruction. Sixth, the CPU is configured to run the original instruction in the context of the production virtual machine in single-step mode. Afterward, the CPU traps

[1] The approach of manipulating the page tables uses the altp2m feature of Xen [10]. This approach does not modify the original memory page. Instead, it creates a new memory page that includes the software breakpoint. By switching the content in the memory tables it can activate/deactivate the breakpoint.

again to the hypervisor. Seventh, steps four and five are repeated with the difference that the VMI application inserts the original breakpoint. Eight, the VMI application tells the hypervisor to resume the production virtual machine.

In total, this procedure requires switching twice between the monitoring virtual machine and the production virtual machine. First, to handle the breakpoint, and second, to reinsert it after the original instruction is executed in single-step mode. Since a software breakpoint actually traps first to the Xen hypervisor and is then handled by a monitoring virtual machine, eight VM Exit/Entry-based context switches are required to handle a single breakpoint (see Fig. 1).

Also, since the VMI application is usually implemented as a userspace application in monitoring virtual machine, additional context switches between the kernel of monitoring virtual machine and the VMI applications are required. The actual number of required context switches depends on many factors, such as the scheduler's decision and how many processes are active in the monitoring virtual machine.

Access to Control Registers. Modifications to the control registers of a CPU can change the general behavior of the system. Hence, observing the changes is a valuable mechanism of virtual machine introspection. The most common example is the monitoring of modifications of the CR3 register. In the Intel architecture, the CR3 register holds a pointer to the directory table base (DTB) of a process. Whenever a process is dispatched by the scheduler of the operating system, it updates the contents of the CR3 register to point to the DTB of the next process. Thus, monitoring changes of the CR3 register can be used to be informed whenever a new process becomes active.

2.5 System Call Tracing

Software breakpoints can be used to monitor the invocation of system calls. To do so, the breakpoint is set on the first instruction of the system call handler function. Since the operating system kernel is used by all processes, this means that all invocations of system calls of all processes are monitored when a breakpoint is set on a system call handler. Thus, if only the system calls of one process should be monitored, this approach can have unnecessary high overhead.

There are two strategies to obtain the system call invocations of a specific process. The first one is to use post-processing. This means that the monitoring application itself filters the events in order to obtain only those system calls that belong to a process with a specific process identifier (PID). This, of course, does not have any positive impact on the performance at run-time. The second strategy is to additionally monitor modifications of the CR3 register, which indicate that a different process in the production virtual machine is being dispatched. Every time a new process is dispatched, the breakpoints can be inserted/removed from main memory so that the tracing is only active when specific processes are running. Sentanoe et al. [16] call this approach *process-bound monitoring*.

3 Improving the Performance

The methods of synchronous virtual machine introspection have a severe performance impact on the production virtual machine. One big part of that problem is the long chain of subsequent operations that are required to handle a VMI event in the hypervisor and the monitoring virtual machine while the production virtual machine is paused. Those steps are processed for each VMI event, even if the VMI application just disregards the event because it does not match a certain filter.

The processing of a breakpoint in the VMI application can be organized into three processing layers. On the lowest layer, the breakpoint is handled with the steps explained in Sect. 2.4. Then on the layer above, the VMI-events of a breakpoint are filtered and processed in the VMI application, e.g., when only breakpoints of a specific process should be monitored VMI events can be filtered based on the value stored in the CR3 register. On the highest layer, additional data from the production virtual machine, e.g., the parameters passed to a function are extracted.

In the following, we discuss possible approaches that aim at minimizing the VMI performance overhead. All of these approaches achieve this goal by reducing the time needed by the data extraction routine and the time during which the monitored virtual machine is paused.

Pre-filtering of Events: In some cases, only a few of the VMI events are necessary for a specific use case. For example, in the case of process-bound monitoring, many events are generated by changes in the CR3 register. However, only events for changes that either write the value of the DTB of the process to be monitored to CR3 or replace that value with a different value are required to enable/disable the breakpoints. All other CR3-based events (e.g., when a context switch from a not monitored to another not monitored process occurs) don't need to be observed. Nevertheless, in the usual approach to monitor CR3 events, a context switch to the monitoring virtual machine is required in all cases, even if the VMI application simply ignores irrelevant events.

One approach to solve this problem is to filter events at the hypervisor level. For example, CR3 events that are captured because of the production virtual machine changing the register from *oldDTB* to *newDTB* could be forwarded to the userspace VMI application only if either *oldDTB* or *newDTB* match the DTB of the process to be monitored.

Breakpoint Mechanism: A fast and efficient breakpoint mechanism helps to minimize the tracing overhead. Since breakpoints can occur frequently, small optimizations can make a big difference. Hence, the implementation of breakpoints should be as optimized as possible so that it requires as little time as possible.

Moreover, the breakpoint mechanism should be implemented into the hypervisor. This eliminates the need to switch to the monitoring virtual machine after a single-step operation was executed as the restoration of the original instruction is independent of the VMI application logic. So instead of switching to the

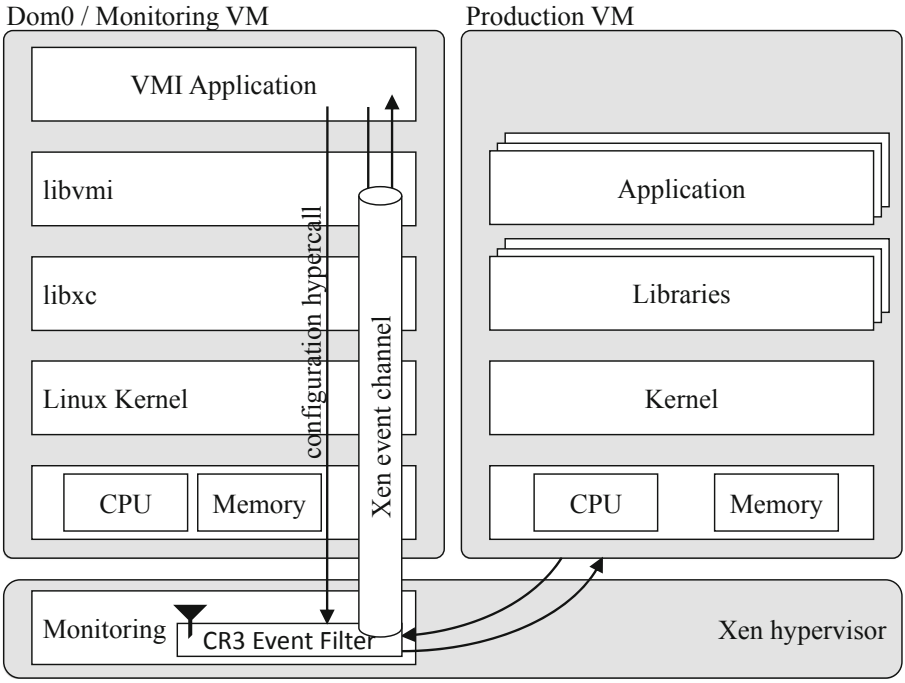

Fig. 2. Our prototype architecture that pre-filters CR3 events in the hypervisor

virtual machine with the VMI application twice, only one context switch to the VMI application is required.

Pre-processing of Events: Another approach to reducing the number of context switches of VMI-based monitoring is to integrate parts of the analysis into the hypervisor so that it is not necessary to always switch to the monitoring virtual machine. However, hypervisors should be designed to be very minimal and contain only the functions to manage virtual machines. Integrating additional VMI code would violate this principle. Thus, we propose to run only very basic information extraction routines that are necessary to access ephemeral data in the context of the hypervisor. Complex analysis of this data should still occur in the context of the monitoring virtual machine. With that approach, it should be possible to minimize the time when production virtual machine must be paused, i.e., the time to extract ephemeral data. The extracted data must then be sent to the VMI-application in the monitoring virtual machine so that it can process the data asynchronous to the execution of the production virtual machine.

4 Prototype

In the following, we describe how the performance impact of process-bound VMI-based monitoring can be minimized by implementing basic pre-filtering

primitives into the hypervisor. The implemented architecture is depicted in Fig. 2. This goal of process-bound monitoring is to decrease the monitoring overhead, especially, when the production virtual machine runs many applications that are not relevant to trace for the analysis. As described in Sect. 2.4, one way to implement process-bound monitoring is to intercept write access to the CR3 register and en-/disable the breakpoints based on the fact which process becomes active. The problem of this approach is that the operating system updates the CR3 registers frequently. Hence, depending on how often the operating system changes the active process, many VMI-events must be processed by the VMI application. However, most of those events are irrelevant, because the VMI-application does not need to be informed about context switches between two processes that are not monitored.

To tackle that problem, our prototype implements a basic pre-filtering mechanism in the Xen hypervisor. The VMI application can configure the filter so that only CR3 events are forwarded to the VMI application when the current or the new value of the CR3 register matches the DTB of the monitored process. The filter can be configured at run-time using hypercalls from the monitoring virtual machine. To achieve that, we extend the hypercall API of Xen to activate or disable the monitoring and set values of CR3 events that should be forwarded. With this approach, we eliminate the need for many context switches to the monitoring virtual machine.

The changes to the Xen hypervisor are essentially the extension of the hypercall and the filtering mechanism. Both modifications require only a few lines of code in the Xen hypervisor. Additionally, the libxc used by the VMI application must be extended to support the introduced parameter of the hypercall. Finally, the VMI application must be extended to use the new CR3 filtering.

5 Evaluation

In this section, we evaluate the performance improvement of our prototype implementation for process-bound monitoring. In our evaluation, we examine whether our prototype that pre-filters VMI events in the hypervisor can help to reduce the performance overhead. For that purpose, we use a similar use case as Sentanoe et al. for the Sarracenia Honeypot [16], which aims to reconstruct the inputs and outputs of an attacker in a bash terminal session. Hence, for this use case, it is sufficient to monitor only the read, write and exec system calls invoked by the main bash process of an SSH session.

We obtain the performance impact of tracing by measuring the time that is consumed by extracting the zip file of the jansson library[2] and compiling the source code. Both the extracting of the zip file and the compilation process require the invocation of many system calls. Most of them are not important for monitoring the attacker's behavior as they belong to the compilation process. Thus, process-bound monitoring of the main bash process that only starts the extraction and compilation should help to reduce the monitoring cost.

[2] https://github.com/akheron/jansson.

Table 1. Measurements of the time, the number of intercepted system calls (read, write, open, close and exec) and CR3 events using four different monitoring mechanisms

Monitoring method	1	2	3	4
Mean time per run [s]	5.25 ± 0.022	148.92 ± 8.66	117.69 ± 8.66	8.31 ± 0.0078
Monitored syscalls	–	1,245,130	4	6
Monitored CR3 events	–	–	20,553,669	276
Overhead	–	2736%	2141%	58%

In our evaluation, we run this use case while monitoring with four different mechanisms:

1. No monitoring.
2. Monitoring the 6system calls of all running processes.
3. Monitoring the system calls of the bash process only. The tracing is based on the fact which process is currently running. The CR3 events are not pre-filtered in the hypervisor. This is similar to the approach of Sentanoe et al.
4. Monitoring the system calls of the bash only by en-/disabling the tracing based on the fact which process is currently running. The CR3 events are pre-filtered in the hypervisor.

The measurements are executed on an HP Elitebook 820 G4 with an Intel(R) Core(TM) i5-7200U CPU @ 2.50 GHz processor. The production virtual machine has one CPU core and 256 MB of main memory. The VMI application is running in the Dom0 and is implemented to monitor the read, write, open, close and exec system call of the bash process of an SSH session that runs the extraction and compilation. The monitoring of the system calls is implemented by placing a breakpoint on the corresponding system call handler function.

The results of our measurements are depicted in Table 1. The second row shows the average execution time of the compilation process that was executed ten times. The execution time is measured in the production virtual machine using the Linux time command. We use the elapsed real-time to quantify the run-time of the unpacking and compilation process over all ten iterations. The third and fourth rows provide the number of monitored system calls and CR3 events over all ten iterations.

At first glance, the approach of process-bound monitoring of system calls appears to be beneficial in order to reduce the impact of monitoring. However, the measurements of monitoring method three show that the required monitoring of write access to the CR3 register in the VMI application almost annihilates this positive effect. This is because the CR3 events are more frequent than system call events. Nevertheless, we measure a slight improvement in performance. The results of the method four show that implementing basic analysis mechanisms into the hypervisor can significantly reduce the impact of VMI-based monitoring. Based on method four we can estimate the overhead for a single CR3 event by dividing the overhead through the number of CR3 events. From this calculation,

we get an overhead of about 11 ms per CR3 event. This long time period can be explained with the different layers in which the event is processed (see Fig. 1).

To sum up, the process of pre-filtering CR3 events decreases the run-time of the execution with process bound tracing from 148.92 s to 8.31 s, which is about 18 times faster. Additionally, these measurements show that integrating basic VMI primitives into the hypervisor can reduce the performance overhead and help to make VMI applicable in production environments. Hence, there is ample room for performance improvement in the current Xen mechanisms for VMI that needs to be addressed by future research. To further improve the performance of synchronous VMI-based monitoring, we, therefore, suggest that more VMI monitoring primitives should be implemented in the Xen hypervisor. Nevertheless, by adding VMI-based functionality to the hypervisor, an additional access control mechanism should be added to the hypervisor. Additionally, if code is loaded to the hypervisor, it must be verified that it does not affect the reliability and security of the overall system.

6 Related Work

The field of dynamic control flow instrumentation has a long history and in the last decades has been many different approaches that tackle the performance overhead. In the following, we want to discuss the most important related approaches. DTrace [3] is a tracing framework for the Solaris operating system. One of the main objectives of DTrace is to have no impact on the performance when the tracing is not implemented. This is implemented by inserting no-operation (nop) instruction at possible probing points. When the monitoring is activated, the nop instructions are replaced by jumps to the analysis routines. The monitoring actions can be implemented by users in the D programming language.

Our prototype approach follows the same approach as the eBPF filters in the Linux kernel [4,12], which have a similar concept as DTrace. The core concept of eBPF is to run application-specific tracing code in the high privileged Linux kernel in order to monitor functions in user and kernel space. This approach helps to minimize (synchronous) context switches to the user space that stalls the execution. The features of eBPF filters are very advanced and the Linux kernel is already able to run small programs, whereas our prototype for VMI-based tracing currently only allows pre-filtering tracing events in the Xen hypervisor.

Tuzel et al. [19] analyzed the performance impact of VMI-based monitoring techniques in detail. The focus of their paper is to analyze to which extent VMI-based monitoring techniques can be detected from the production virtual machine. The result of their conducted study is that VMI-based monitoring in their setup is not stealthy to applications running in the production virtual machine. The focus of our paper is to decrease the monitoring overhead to make it applicable in production environments but not to make VMI-based monitoring completely stealthy.

Klemperer [9] proposed to use copy-on-write-based snapshots in order to perform the analysis on a non-changing system state. For this purpose, they

extended the KVM hypervisor and hook instructions that alter memory pages during the normal execution. If a page is modified they copy the original version of the page for the snapshot. After the analysis is finished, the snapshot gets deleted and changes to memory pages are not monitored anymore. This approach is probably most effective when larger parts of main memory must be analyzed. For the extraction of function call parameters when a breakpoint is invoked this approach is too expensive.

Drakvuf [11] is a VMI framework that uses libvmi and Xen and is mainly designed for dynamic analysis of Windows malware. It uses the altp2m approach [10] to implement software breakpoints. The advantage of this approach is that instead of replacing the original instruction with the INT3 instruction in memory, it creates a new memory page with the software breakpoint. Depending on whether a process in the production virtual machine is reading from that page or executing an instruction the memory page is swapped in the page tables. The rVMI framework designed by Pfoh and Vogl [15] is similar to Drakvuf. However, it uses a patched version of KVM and Qemu instead of Xen[3]. The communication between the VMI application and Qemu is established via the QMP interface of Qemu.

Westphal et al. [21] define a VMI monitoring language that supports the most common methods for VMI. Their prototype is implemented for the VMware KVM hypervisor. Similarly to our proposed approach, they execute VMI analysis scripts in the domain of the KVM hypervisor.

Bushouse et al. [2] implement VMI-based monitoring into Linux containers running in the Dom0 of Xen instead of running the VMI-application in a monitoring virtual machine. This should not have any positive beneficial impact on the performance.

7 Conclusions

By implementing a pre-filtering approach in the hypervisor, we significantly reduce the impact of VMI-based monitoring on the production virtual machine. Our prototype reduces the monitoring overhead for process-bound monitoring by a factor of 18. We, therefore, conclude that more complex event processing in the hypervisor, such as the implementation of a breakpoint mechanism, can help to solve the problem of minimizing the performance impact of VMI-based tracing.

Acknowledgment. This work has been supported by the German Research Foundation (DFG) in the project ARADIA (RE 3590/3-1).

[3] Currently, the KVM hypervisor does not support synchronous VMI-based monitoring out of the box. To use synchronous VMI-based monitoring mechanisms, the KVM hypervisor must be patched.

References

1. Barham, P., et al.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. **37**(5), 164–177 (2003)
2. Bushouse, M., Reeves, D.: Furnace: self-service tenant VMI for the cloud. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) RAID 2018. LNCS, vol. 11050, pp. 647–669. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_30
3. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 2004, pp. 15–28. USENIX Association, Berkeley (2004). http://dl.acm.org/citation.cfm?id=1247415.1247417
4. Fleming, M.: A thorough introduction to eBPF (2017). https://lwn.net/Articles/740157/. Accessed 26 Sept 2019
5. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed Systems Security Symposium, pp. 191–206 (2003)
6. Hebbal, Y., Laniepce, S., Menaud, J.M.: Virtual machine introspection: techniques and applications. In: 2015 10th International Conference on Availability, Reliability and Security, pp. 676–685, August 2015. https://doi.org/10.1109/ARES.2015.43
7. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual (2017). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
8. Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R.: SoK: introspections on trust and the semantic gap. In: IEEE Symposium on Security and Privacy, pp. 605–620 (2014)
9. Klemperer, P., Jeon, H.Y., Payne, B.D., Hoe, J.C.: High-performance memory snapshotting for real-time, consistent, hypervisor-based monitors. IEEE Trans. Depend. Secur. Comput., 1 (2018). https://doi.org/10.1109/TDSC.2018.2805904
10. Lengyel, T.K.: Stealthy monitoring with Xen altp2m (2016). https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/. Accessed 31 Jan 2019
11. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 386–395 (2014)
12. McCanne, S., Jacobson, V.: The BSD packet filter: a new architecture for user-level packet capture. In: Proceedings of the USENIX Winter 1993 Conference, USENIX 1993, p. 2. USENIX Association, Berkeley (1993). http://dl.acm.org/citation.cfm?id=1267303.1267305
13. Payne, B.D.: Simplifying virtual machine introspection using LibVMI. Sandia Report, pp. 43–44 (2012). http://libvmi.com/
14. Payne, B.D., de A. Carbone, M.D.P., Lee, W.: Secure and flexible monitoring of virtual machines. In: 23rd Annual Computer Security Applications Conference (ACSAC 2007), pp. 385–397, December 2007. https://doi.org/10.1109/ACSAC.2007.10
15. Pfoh, J., Vogl, S.: rVMI - a new Paradigm for Full System Analysis (2017). https://github.com/fireeye/rvmi. Accessed 31 Jan 2019

16. Sentanoe, S., Taubmann, B., Reiser, H.P.: *Sarracenia*: enhancing the performance and stealthiness of SSH honeypots using virtual machine introspection. In: Gruschka, N. (ed.) NordSec 2018. LNCS, vol. 11252, pp. 255–271. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03638-6_16
17. Shi, J., Yang, Y., Li, C.: A disjunctive VMI model based on XSM. In: 2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity), pp. 921–925, December 2015. https://doi.org/10.1109/SmartCity.2015.188
18. Taubmann, B., Rakotondravony, N., Reiser, H.P.: CloudPhylactor: harnessing mandatory access control for virtual machine introspection in cloud data centers. In: The 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2016). IEEE (2016)
19. Tuzel, T., Bridgman, M., Zepf, J., Lengyel, T.K., Temkin, K.: Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. Digit. Invest. **26**, S98–S106 (2018). https://doi.org/10.1016/j.diin.2018.04.015
20. Uhlig, R., et al.: Intel virtualization technology. Computer **38**(5), 48–56 (2005). https://doi.org/10.1109/MC.2005.163
21. Westphal, F., Axelsson, S., Neuhaus, C., Polze, A.: VMI-PL: a monitoring language for virtual platforms using virtual machine introspection. Digit. Invest. **11**, S85–S94 (2014). https://doi.org/10.1016/j.diin.2014.05.016. http://www.sciencedirect.com/science/article/pii/S1742287614000590, fourteenth Annual DFRWS Conference