VMI-PL: A Monitoring Language for Virtual Platforms Using Virtual Machine Introspection

Florian Westphal[a,b], Stefan Axelsson[a], Christian Neuhaus[b], Andreas Polze[b]

[a] *Blekinge Institute of Technology, Sweden*
[b] *Hasso-Plattner-Institute, University of Potsdam, Germany*

Abstract

With the growth of virtualization and cloud computing, more and more forensic investigations rely on being able to perform live forensics on a virtual machine using virtual machine introspection (VMI). Inspecting a virtual machine through its hypervisor enables investigation without risking contamination of the evidence, crashing the computer, etc.

To further access to these techniques for the investigator/researcher we have developed a new VMI monitoring language. This language is based on a review of the most commonly used VMI-techniques to date, and it enables the user to monitor the virtual machine's memory, events and data streams. A prototype implementation of our monitoring system was implemented in KVM, though implementation on any hypervisor that uses the common x86 virtualization hardware assistance support should be straightforward. Our prototype outperforms the proprietary VMWare VProbes in many cases, with a maximum performance loss of 18% for a realistic test case, which we consider acceptable. Our implementation is freely available under a liberal software distribution license.

1. Introduction

More and more computing services are being provided remotely in the form of so called "cloud based services" in the form of *virtual* machines, as it is done by Windows Azure, or Amazon EC2, etc. Thus virtualization is becoming an evermore important tool for providing computing services to customers, especially when it comes to providing e.g. web servers, data base servers etc. so called *infrastructure as a service (IaaS)*. This market segment is growing fast, at present.

Hence, the probability that forensic investigations will have to be performed on virtual machines will continue to increase. Whereas virtual machines can be treated and analyzed as normal computers [1], their use in cloud settings poses several challenges, and also opportunities for the investigation. These challenges include legal issues, such as cross-border legislation problems [2], as well as trust issues [3].

Despite these challenges, virtual machines also have an advantage over real computers when it comes to live forensics. Whereas live forensics on a real machine always causes state changes, possibly contaminating the evidence [4], live forensics on a virtual machine can be done with minimal interference with its state, by using virtual machine introspection (VMI) [5]. Several VMI techniques have been developed and implemented, in recent years.

To this end we have developed a domain specific language (DSL) named VMI-PL, that can be used to define and apply common VMI techniques for the monitoring of virtual machines. This DSL provides a simple way for the user to specify virtual machine events which should be intercepted, registers and virtual memory, which should be read or written to, and data streams from and to the virtual machine, which should be logged. In addition to data about the hardware level, the language provides means to obtain operating system level information by interpreting the data available to the hypervisor. This combination of ease of use and capabilities it provides makes it, in our opinion, a valuable tool for live forensic investigations.

In connection with the language design, we propose a new classification for VMI techniques, which is based on the respective key mechanism of the VMI technique. These key mechanisms can be data accesses, hardware events, or data transfers across the border between virtual machine and hypervisor. In addition to these theoretical contributions, we also developed a prototype, implementing our language design in the Kernel-based Virtual Machine (KVM) hypervisor, which is freely available under an open source license.

2. Related Work

As VMI-PL allows the implementation of new monitoring strategies without modifying the hypervisor further beyond that which was necessary to implement support for VMI-PL. We limit the related work section to those approaches that share this characteristic. This excludes approaches such as HyperDbg [6] (which is not a true hypervisor based system), and VMST [7] (that while their approach has much to commend it, is limited to just the one read-only strategy of memory introspection). We have

only found two other publicly available solutions for which this is true: LibVMI[1], and VMware VProbes [8].

LibVMI is an introspection library for Xen and KVM, which was derived from the XenAccess project [9]. While Xen is fully supported by this library, the support for KVM is more limited, due to missing hooks in the KVM code. With LibVMI, it is possible to read and write virtual memory, as well as to register handlers for certain events occurring within a virtual machine. These events can be memory accesses, register writes, or instruction execution, where instruction execution can be tracked either in single stepping mode or by specifying a particular virtual address.

VProbes, on the other hand, is a monitoring solution for VMware Workstation and VMware Fusion, which provides the possibility to specify probes, which are executed on certain events. The specification of these probes is done in VMware's own language called *Emmett*. While the initial idea of VProbes was to support external debugging of virtual machines [10], it has also been shown to be useful in the context of e.g. intrusion detection [11]. It should be noted that *Emmett* only allows the reading of information from the monitored host.

In contrast to VProbes, our monitoring language supports writing to effect the manipulation of the internal state of the virtual machine.

Our monitoring language differs from both these solutions in that it provides access to operating system level information and events. This simplifies investigations, since the interpretation of the hardware state is done automatically. Another benefit from our approach is that it monitors data streams from and to the virtual machine, which cannot be monitored using the other solutions. This feature enables an investigator to monitor network traffic, as well as the keyboard input of a user. Finally, our monitoring language has a simple syntax, which enables a user to obtain data from a virtual machine without the need to have a extensive background in low level programming.

3. Virtual Machine Introspection

Virtual machine introspection (VMI) was coined by Garfinkel and Rosenblum [12] and was defined by Pfoh et al. as a "*method of monitoring and analyzing the state of a virtual machine from the hypervisor level.*" [13]. Since VMI obtains this state information directly from the hypervisor, the semantic knowledge of the guest operating system's abstractions is lost. Hence, it is difficult to interpret the data. This was first described by Chen and Noble as the *semantic gap* problem [14]. In order to overcome the semantic gap, there are different strategies, which were categorized by Pfoh et al. [13] as *out-of-band delivery*—where the needed semantic knowledge is supplied externally, *in-band delivery*—that pass the needed data along with the semantic information directly from inside the guest operating system to the hypervisor, and *derivation*—that derives the needed information from how the guest operating system is using the given hardware interface.

These approaches have different advantages and disadvantages when it comes to the amount of information they can easily provide and how difficult it is for an attacker to deceive them. Additionally, it should be noted that *in-band delivery* techniques are less suitable for live forensics, since they require the running of software inside the virtual machine, potentially manipulating its state to a considerable extent, and therefore neglecting the advantage VMI normally provides for live forensics.

4. VMI Techniques

For reasons that will become clear we decided to further refine this classification of VMI techniques.

While all techniques described in the following sections can be categorized as *out-of-band delivery* or *derivation* techniques, we will divide them further into *data-based*, *event-based*, and *stream-based* techniques. This categorization has been useful in the specification of our monitoring language, since it takes into account the underlying mechanisms of the particular VMI technique.

4.1. Data-Based Techniques

Data-based VMI techniques focus on the virtual storage of a virtual machine, such as registers, main memory, and the virtual hard disk. These techniques obtain information from the virtual machine by reading that storage and require external knowledge to interpret the information. Therefore, these techniques belong to the *out-of-band delivery* techniques and can be defined as follows:

Definition 1 *Data-Based Techniques are out-of-band delivery techniques, which analyze data from virtual storage, in order to gain knowledge about information processed within the guest operating system of a virtual machine.*

Based on this definition, VMI techniques in this category are *register introspection*, *main memory introspection*, and *virtual disk introspection*. While disk introspection can be done during a "dead" analysis, register and memory introspection can only be performed during a live analysis on a running virtual machine, due to their volatility.

In general, memory introspection can provide the same information as a forensic memory analysis (FMA) performed on a dump of physical memory, but also faces the same challenge, namely the semantic gap [15]. Information, which can be obtained include, for instance, a list of running processes, loaded kernel modules, or open sockets [16], as well as program specific information, such as open files or encryption keys [17]. The latter are of course a common target in a majority of criminal investigations that target suspects using a laptop, or similar, in a home or small business setting. However, we venture to guess

[1] https://code.google.com/p/vmitools/

that for the virtual target, in-memory root kits, viruses/-worms and the like are more often of more interest, as the relative proportion of encrypted storage is lower in a server setting.

Even though both approaches can obtain the same information, memory introspection has an advantage over FMA, in that it is in general easier and less intrusive to either dump the whole virtual memory or just read out the needed parts, since the hypervisor can provide direct access to the virtual memory. In contrast, dumping the main memory of a physical machine usually requires the installation of specialized programs to perform this task. This changes the state of the computer, and also runs the real risk of freezing the computer or not successfully acquiring all of the memory. This is especially true on a compromised target, as the compromise may make the returned information unreliable.

4.2. Event-Based Techniques

Event-based techniques intercept transitions from the virtual machine to the hypervisor, in order to gain knowledge about the actions of the operating system. These transitions occur when the guest operating system executes certain sensitive instructions. Since these instructions affect the whole system state, they have an impact on the host machine as well, as they compromise the isolation between virtual machine and host. For this reason, all of these sensitive instructions have to cause a transition to the hypervisor, so that the hypervisor can emulate them without influencing the host system. This sufficient condition for virtualization was first formalized by Popek and Goldberg in 1974 [18].

Since the sensitive instructions have to be emulated by the hypervisor, they can be easily intercepted and certain information can be derived from their occurrence. Due to the fact that information is obtained by monitoring how the guest operating system uses the provided hardware interface, these techniques are *derivation* techniques, defined as follows:

Definition 2 *Event-Based Techniques are derivation techniques, which make use of transitions from the virtual machine to the hypervisor, in order to obtain knowledge about the actions of a virtualized operating system by interpreting the cause of those transitions.*

One commonly used event-based technique is to monitor writes to the `CR3` register [19]. (For a very nice in-depth overview of how to derive VMI state information on the x86 architecture, see Pfho et. al [20].) Due to the fact that this write changes the virtual address space mapping, it is a sensitive instruction when hardware support for memory virtualization, known as extended page tables (EPT) or nested paging, is not used, due to the hypervisor having to do the memory management for the virtual machine then. Monitoring the change of virtual address spaces is instructive, due to the fact that it indicates that either a new user mode process was started or that a context switch from one user mode process to another occurred.

In addition to sensitive instructions, interrupts can also cause a transition to the hypervisor and can therefore be intercepted. The interrupts which are most often monitored, are the user defined interrupt used for issuing system calls, page faults, debug exceptions, and invalid opcode exceptions.

Monitoring page faults, on the other hand, provides the possibility to monitor data as soon as it is loaded in main memory or to detect access violations [21]. Furthermore, it is also possible to introduce a write or execution protection for certain memory pages by setting them not writable or not executable in the corresponding page table entries. Using these types of protection, it is possible to detect code injection [22] or stack overflow exploits as they happen.

4.3. Stream-Based Techniques

Another way to obtain information from a virtual machine is to monitor the data streams flowing in and out of the virtual machine. Since the interpretation of the data in those streams requires external information, stream-based techniques belong to the category of *out-of-band delivery* techniques:

Definition 3 *Stream-Based Techniques are out-of-band delivery techniques, that process input and output data to/from a virtual machine. This data is intercepted as it crosses the border between virtual machine and hypervisor by passing through a virtualized device.*

The most commonly used stream-based technique is network introspection. This can be done by intercepting the network traffic to and from the virtual machine directly at the virtual network interface [23]. By monitoring the network traffic at this point, it is impossible for potential rootkits to conceal open network connections. Furthermore, by coupling this technique with other VMI techniques, it is possible to correlate network traffic with running processes, for example identifying a process behind known malicious network traffic. This correlation can be done by identifying which process was executing while certain network traffic was sent. Which user mode process was running can be determined by monitoring the `CR3` register, as described before.

In addition to network introspection, it is also possible to log the keyboard input by intercepting input at the virtual keyboard controller [24, 6]. This provides the possibility to closely monitor the actions of a user on the monitored virtual machine.

5. VMI-PL Specification

The purpose of the proposed Virtual Machine Introspection Probe Language (VMI-PL) is to provide a simple way to obtain the relevant information from a virtual machine without having to implement the details of the necessary VMI techniques. With VMI-PL, a user can write

scripts to specify probes to obtain the needed data. These probes represent common VMI techniques and are defined as follows:

Definition 4 *A probe is a specific action, which is executed when certain preconditions are met. This action can be used either to gather information from a virtual machine or to manipulate its state.*

Since VMI-PL probes represent certain VMI techniques, there are three different probe types each corresponding to the three different technique categories.

In keeping with our definitions above, three types of probes are implemented in VMI-PL. These probe types are *data probes*, *event probes*, and *stream probes*. As mentioned in the probe definition, specified probes are executed when their preconditions are met. While the precondition for event type probes is the occurrence of the corresponding transition to the hypervisor. Data probes need a specific trigger to determine when data should be read from the virtual machine. In VMI-PL, such a trigger would either be the execution of an event probe or when a filter condition is met. The different triggers for data probes are mirrored in the VMI-PL syntax, as shown in Listing 1. Here, one can see that filter conditions are expressed by special instructions, which are described later. Data probes and event probes are closely coupled, since data probes need event probes as triggers, and event probes need data probes to generate output. This it not the case for stream probes, which can be used alone, as the trigger for a stream probe, as well as its output, is defined by the corresponding data stream.

It is possible to place any number of data probes within an event probe or filter. Other probe types cannot be nested, since that wouldn't make sense semantically. For example, is it impossible for an event to occur while another event is already handled, since the monitored operating system is already stopped and not executing when the first even has triggered.

```
Configuration{
        <configuration-item>
}
<event-based-probe>(<parameters>){
        <data-based-probe>(<parameters>)
        <filter>(<parameters>){
                <data-based-probe>(<parameters>)
        }
}
<stream-based-probe>(<parameters>)
```

Listing 1: VMI-PL Script Structure

Since the data exchanged via monitored streams have a standardized format, stream probes do not face the semantic gap problem. Data and event probes, on the other hand do, since they obtain data directly from the virtualized hardware. Therefore, external knowledge about either the monitored guest operating system or about the hardware interface used is required to interpret the retrieved data. Since the needed external knowledge is different for each probe, we will describe it along with the probes. In VMI-PL, this interpretation can be done either by the user, by retrieving the raw data available to the virtual hardware, or directly by means provided by VMI-PL.

Due to these two different possibilities, the available probes can be divided into two different categories: *hardware-level probes* and *operating-system-level probes*. Hardware-level probes gather data as seen by the hardware without any semantic information, and operating-system-level probes gather information by using external knowledge about the operating system. While there is at least one approach which free users from the need to provide such external knowledge themselves, instead automating the process, [7], we prefer our approach since it leads to better performance. Also, our mehtod does not require the running of additional virtual machines with a copies of each monitored operating system, in addition the systems that are already being monitored. Therefore, the external knowledge which is necessary for those probes has to be provided by the user as configuration items in the configuration section of a VMI-PL script. This configuration makes it easy to support new operating systems and kernel versions, as long as their implementation is similar to other commonly used operating systems, such as Microsoft Windows or Linux.

In addition to the three mentioned probe types, VMI-PL also provides filters to define when data should be read, and reconfiguration instructions to adjust the way in which a specific virtual machine is monitored. Apart from these language constructs, VMI-PL does not provide any other constructs, such as variables or loops. VMI-PL is only meant to provide the possibility to apply VMI techniques to acquire data from a virtual machine, the processing of that data can then be handled by any suitable software. The advantage of a specification language that doesn't contain loops, recursion etc. to support general computation and data processing, is that the execution of the probes will always terminate in a finite amount of time. The one probe type that traverses lists supports a configurable upper transition bound, to stop execution when faced with e.g. loops in the list structure. This means we can guarantee that the virtual machine execution does not get stuck in the hypervisor.

The following sections describe the language constructs in more detail. Due to a lack of space, not all probes are presented here. Only a few representative probes are shown. The complete list, and description, of all probes can be found in [25].

5.1. Data Probes

VMI-PL's data probes make it possible to retrieve data from a virtual machine, as well as manipulate data inside it. Whenever a data probe obtains data from the virtual machine, the last parameter to this probe is used to specify where data should be sent. It is possible to specify two different output targets, either a file or a data stream, which can be read by another program. In order to direct the output of a data probe to a file, the full path to the file

has to be provided. To use data streams, a stream identifier has to be provided, which has to start with a "#". This identifier is used to provide the connection information, when the VMI-PL script is executed.

The VMI-PL specification contains several hardware-level data probes to read out registers or main memory, or write to them. The difference between these probes is how the particular part of memory is specified. The `ReadMemory` data probe, for instance, takes a virtual address and the amount of bytes to be read as parameters. When the probe is evaluated, it reads from the given virtual address as determined by the mapping that is valid when the probe triggers, and sends the result to the specified output target. One example of a memory manipulating data probe is `WriteToMemoryAt`. This probe takes a register name, an offset value, and the value which should be written to memory, as parameters. On evaluation, the virtual address, which the value should be written to, is determined based on the content of the specified register and the offset.

An example of an operating-system-level data probe is the `ProcessList` probe. This probe uses external knowledge, provided in the configuration section, to read out the operating system's data structures containing information about currently running processes. This probe can be configured to output the process identifier (`PID`), the process' name (`NAME`), the path to the corresponding executable (`PATH`), and the address of the process' page global directory (`PGDP`) of each process by providing the respective field names as parameters. As mentioned before, since this probe is an operating-system-level probe, it requires external knowledge to bridge the semantic gap. This knowledge includes the location of the first process list element in main memory, as well as the necessary offsets within this element to read out the requested information and iterate over the process list which we assume to be a linked list. This is the case in all major operating systems that are typically run under a an x86 hypervisor, such as Microsoft Windows and Linux. (If the process list is a table, normal data probing can be used).

We are currently working on generalising this probe type to support recursion over more general list structures.

5.2. Event Probes

As one can see from Listing 1, event probe definitions contain data probes and filters, which are evaluated when the event specified by the probe takes place. Like data probes, event probes can be divided into the two categories hardware-level probes and operating-system-level probes. Examples of hardware-level probes are the `CRWrite`, and `ExecuteAt` probes. The `CRWrite` event probe expects the number of one of the control registers as a parameter. Its attached data probes and filters are then evaluated as soon as a write to the specified control register takes place. This can be used, for instance, to determine which process was scheduled for execution, when the `CR3` register is monitored with this probe. The `ExecuteAt` event probe, on the other hand, takes a virtual address as a parameter and is triggered when the code at this address is executed.

As mentioned before, the VMI-PL specification also contains operating-system-level probes, such as the `ProcessStart` and `Syscall` probe. The `ProcessStart` event probe is executed when either one specific process or an arbitrary process is started. A process can be specified by name or executable path. Due to the fact that this specification requires the same information as the `ProcessList` data probe provides, the external knowledge needed is the same. In addition to the starting of processes it is also possible to monitor when a process is scheduled for execution and when a process is terminated. The `Syscall` probe is triggered when the guest operating system issues either a specific system call or an arbitrary system call, depending on the parameter passed to the probe. Normally, it should be enough to provide this probe with knowledge about which user defined interrupt is used to issue system calls and which register is supposed to contain the system call number. Nevertheless, under certain circumstances, which are described in section 6, it is also necessary to specify the virtual address of the system call dispatcher in the configuration section.

5.3. Stream Probes

Similar to data probes, all stream probes take an output specification, containing either a file path or a data stream identifier as the last parameter. The captured stream data is then directed to this output. As mentioned before, stream probes are not divided into the same two categories that data and event probes are. Here, we will exemplify with the `CaptureNetwork` and `CaptureKeyboardInput` stream probes.

The `CaptureNetwork` probe saves captured network traffic to its output target. In order to specify from which virtual network interface the network traffic should be intercepted, this probe takes the interface's MAC address as a parameter. The `CaptureKeyboardInput` probe does not take any other parameters and outputs the observed keystrokes to the specified output.

5.4. Filters

Filters can be used within an event probe to add additional preconditions to the evaluation of data probes. As described before, data probes which are placed within a filter, are only executed when the filter condition is met. Filter conditions can for example be the presence of a certain value in a specific register or a specific memory address. Hence, the defined filters are: `RegisterHasValue`, `ValueAtAddressIs`, and `ValueAtRegisterIs`.

While the `RegisterHasValue` filter takes a register name and a value to compare the register's content with the given value, the `ValueAtAddressIs` filter does the same for a given virtual address. Lastly, the `ValueAtRegisterIs` filter calculates the virtual address based on the value

read from the given register and the given offset first, before comparing the value at this address with the given one.

5.5. Reconfiguration Instructions

The reconfiguration option of VMI-PL makes it possible to adjust how the virtual machine is monitored. By using this option, an external program can analyze the data from a virtual machine and change the way it is monitored. In order to achieve this, a new VMI-PL script can be given to the VMI-PL execution environment at any time, as long as the monitored virtual machine is running. Such a script can contain new probe definitions as well as reconfiguration instructions to modify already existing ones. The instructions, which were introduced to support reconfiguration are `Pause`, `Reconfigure`, and `Deregister`.

Similar to a data probe, the `Pause` instruction can be used within an event probe or within a filter to cause the virtual machine to suspend, as soon as the corresponding event has taken place or the respective filter condition is met. This can be used to gain sufficient time to analyze the monitored information and reconfigure accordingly.

Event probes that are already defined can be reconfigured using the `Reconfigure` instruction. This instruction takes as parameter the event probe identifier, which is assigned to every event probe by the VMI-PL execution environment. With this instruction, one can redefine which data probes and filters should be executed, when the corresponding event occurs. The general syntax of this instruction is shown in Listing 2.

```
Reconfigure(<event probe id>){
        <data-based-probe>(<parameters>)
        <filter>(<parameters>){
                <data-based-probe>(<parameters>)
        }
}
```

Listing 2: VMI-PL `Reconfigure`

Additionally, it is also possible to deregister previously defined event and stream probes by providing their respective identifier to the `Deregister` instruction.

5.6. Examples

An example application of VMI-PL is process life cycle monitoring, as shown in Listing 3. This VMI-PL application logs the currently running process by intercepting writes to the `CR3` register and logs when a process is terminated. The process termination is determined by monitoring when the Linux's kernel `do_exit` function at the specified virtual address is executed. This monitoring application can be used, for instance, to uncover hidden processes. Even if a process is not contained in the operating system's process list, for example due to direct kernel object manipulation (DKOM), its virtual address space has to be set up before it can be executed and hence will be detected by us.

```
CRWrite(3){
        ReadRegister(CR3, /tmp/schedules_processes)
}
ExecuteAt(0xc104f060){
        ReadRegister(CR3, /tmp/terminated_processes)
}
```

Listing 3: VMI-PL Script for Process Life Cycle Monitoring

Another example VMI-PL application is process execution monitoring. This logs invoked system calls using the `Syscall` event probe, as shown in Listing 4. As described before, the `Syscall` event probe requires external knowledge, such as the address of the system call dispatcher, which can be provided in the configuration section of the VMI-PL script.

```
Configuration{
        SyscallDispatcherAddress: 0xc15780e8
        SyscallNumberLocation: EAX
        SyscallInterruptNumber: 128
}
Syscall{
        ReadRegister(EAX, #syscall_monitor)
        ReadRegister(CR3, #syscall_monitor)
}
```

Listing 4: VMI-PL Script for Process Execution Monitoring

6. Implementation

We implemented a prototype VMI-PL by patching the KVM hypervisor. Our architecture consists of two parts, as shown in Figure 1: the VMI-PL execution environment which consists of roughly 1500 lines of Python, and the modified version of the KVM kernel module which weighs in at circa 1500 lines of C (most of which is additions with only a few lines of hooking into existing code required).

Execution Environment. The execution environment is a user mode program creating the bridge between the user and the hypervisor. It parses the provided VMI-PL script and transfers the configuration to the hypervisor. During the parsing of the VMI-PL script, the execution environment assigns identifiers to the different event and stream probes and returns these identifiers to the user. They can later be used for reconfiguring the virtual machine monitoring.

The execution environment also maps the specified output targets, i.e., files or data streams, to corresponding connection information, which are returned to the user, in case the specified target was a data stream. In case of a file target, the execution environment sets up a listener with the necessary connection information to receive the data sent from the hypervisor and store it in the specified file. In addition, the execution environment takes care of the reconfiguration of probes by parsing the updated VMI-PL script and transferring the updated information to the hypervisor.

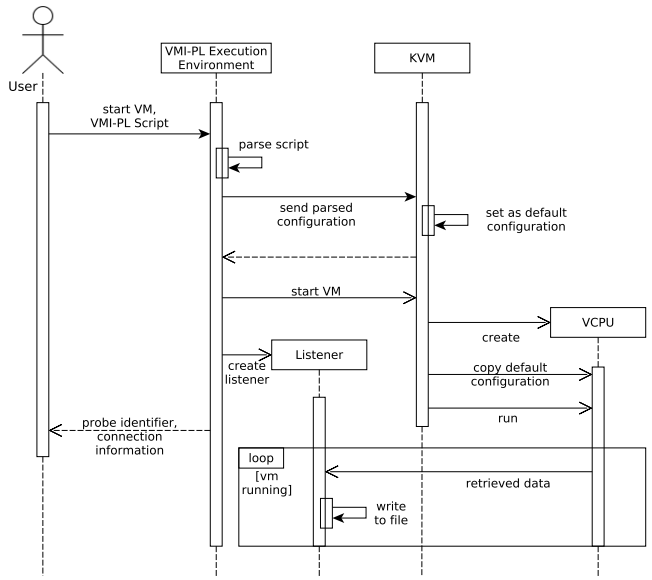

Figure 1: Interaction between Different Elements of the VMI-PL Implementation

KVM. The KVM kernel module was modified to provide a communication interface to the execution environment to receive the probe configurations. Additionally, the creation of virtual processors was extended, in order to attach these configurations to the newly created processor. Such a configuration contains external knowledge for operating-system-level probes, as well as information about active event probes, their parameters, and attached data probes and filters. This configuration is consulted by the added monitoring hooks every time a transition to the hypervisor occurs. When an event probe corresponding to the monitored transition is found to be active, all its attached data probes and filters are evaluated and the data obtained by the data probes is broadcast to all listeners. Since the corresponding configuration is attached to each created virtual processor, VMI-PL makes it possible to monitor several virtual machines, running different operating systems, at the same time.

Communication. The communication between the KVM kernel module and the execution environment, and other user mode programs, is implemented using Linux's netlink sockets. While the execution environment sends the respective configuration to the hypervisor using unicast messages, the hypervisor sends all received data as broadcast messages. This makes it possible for any user mode program to receive and analyze the logged data, as long as it has knowledge of the corresponding connection information. This connection information include the protocol type and the group identifier. While the protocol type is unique per virtual machine, the broadcast group identifier is used to direct the obtained data to different output targets.

Data Probes. The VMI-PL data probes are implemented as functions in the KVM kernel module, which are executed as soon as an event probe or filter with the corresponding data probe is evaluated. In order to access registers, translate virtual memory addresses, and read and write the virtual memory, those functions in turn make use of KVM's internal address translation functions. Hence, address translation is always performed in the context of the current running process in the monitored OS. While this is no problem when kernel data is accessed, since the kernel address space is normally mapped in every process address space, process specific data has to be handled differently. To retrieve data from one specific process, one could use a filter based on the contents of the `CR3` register. This makes it possible to configure memory reads for a specific process. Since a process' PGD address can only be determined at runtime, these filters have to be registered using the reconfiguration capabilities of VMI-PL. After obtaining the requested data, our implementation broadcasts it using the virtual processor's netlink socket and the group identifier, which was configured for this data probe.

Event Probes. As mentioned earlier, event probes are implemented by hooks placed in the KVM transition handling code, which check on every transition if the corresponding probe was configured. For instance, writes to the `CR3` register are intercepted in KVM's handler for writes to the control registers. If the corresponding hook finds the event probe configured, the attached data probes and filters are evaluated.

While event probes, such as `CRWrite` do not require any manipulation of the guest operating system, the event probe implementation has to perform certain alterations of the guest for other probe types, for instance, the `ExecuteAt` event probe. When this probe is configured, the VMI-PL implementation has to set up one of the guest's debug registers to cause a debug exception, when the code at the specified address is executed. The hook for this probe is located in the corresponding KVM handler and evaluates the data probes and filters, if the transition was caused by the correct debug register.

The last event probe implementation, described here, is the implementation of the `Syscall` probe. Since our prototype was developed for Intel processors, our implementation has to deal with the fact that Intel processors do not make it possible to transition to the hypervisor on user defined interrupts. Therefore, we determine the virtual address of the system call dispatcher and replace its first instructions with the undefined instruction `ud2`. This causes a transition to the hypervisor whenever a system call is issued, due to an invalid opcode exception. The exception is then intercepted by the corresponding hook, and the respective event probe can be evaluated. Afterwards,

the replaced instructions have to be emulated in the hypervisor, before the execution of the virtual machine can continue.

Normally, the address for the system call dispatcher can be determined by reading out either the IDT or the IA32_SYSENTER_EIP model specific register (MSR), depending on how system calls are issued by the guest operating system. While writes to the IA32_SYSENTER_EIP MSR can cause a transition to the hypervisor, writes to the IDTR, which would indicate when the IDT was initialized, only cause transitions on newer Intel processors. This makes it necessary to provide the address to the system call dispatcher directly for older processors.

It should be noted that the `ud2` instruction that we write could be detected by malware, which could then take evasive action. This is difficult to counter with OSes that uses interrupt table based system calls (like Linux int 0x80), but for SYSENTER based system calls there are ways to make such monitoring invisible [26], which we plan to implement.

Stream Probes. Even though the implementation of the stream probes is straightforward, due to a lack of space we refer to the previously cited source for their description. The only complication is that a different communication structure between these probes and the VMI-PL execution environment is needed. The reason is that the data for these probes are intercepted at the virtualized devices and not, in the KVM kernel module as for data or event probes.

Filters & Reconfiguration Instructions. Filters are simply implemented as functions within the KVM kernel module, which are executed whenever an event probe is triggered to which a filter is attached. These functions read out registers or virtual memory in the same way as data probes and compare the obtained data with their configured values. If the values match, the data probes attached to this filter are evaluated.

The `Reconfigure` and `Deregister` instructions are mainly implemented in the execution environment, which parses these instructions and transmits the configuration changes to the respective virtual processor. The `Pause` instruction, on the other hand, is implemented in the kernel module. Whenever an event probe or filter is evaluated, which has the `Pause` instruction active, the kernel sends a message to the execution environment with the request to suspend the virtual machine. This is done, as soon as the corresponding listener receives this message.

7. Performance Evaluation

We evaluated the performance of our VMI-PL implementation and compared the results with VMware VProbes as these are directly comparable. The following results were obtained on a host with 4 GB of RAM and an Intel Core 2 Duo processor running at 2.53 GHz. The

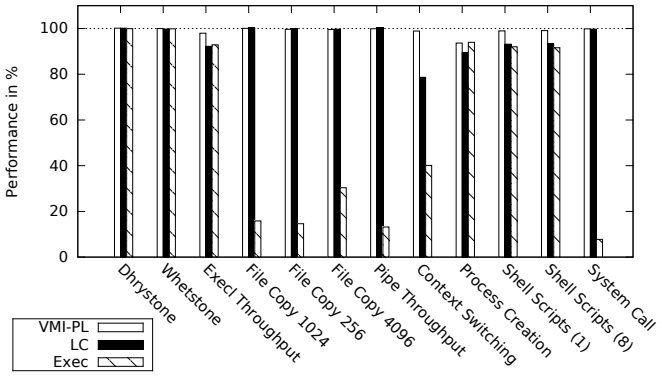

Figure 2: Relative performance impact of VMI-PL on the UnixBench benchmarks

processor supports Intel's hardware-assistance for virtualization (VT-x), but not the extended page table feature. The host operating system was Linux Mint Debian Edition with a 32-bit Linux kernel version 3.2.0. As the hypervisor, we used the KVM external module kit, kvm-kmod-3.10.1, for our VMI-PL implementation and VMware Workstation 10.0.1 for the comparison with VProbes. Under both hypervisors, we set up a virtual machine with one virtual processor and 256 MB of RAM. In both cases, this virtual machine ran Ubuntu 12.04 with a 32-bit Linux kernel version 3.2.0.

For our performance evaluation, we used the UnixBench benchmark suite version 5.1.3. This benchmark suite was executed on the guest operating system in four different scenarios. In the first scenario there was no support for VMI-PL or VProbes configured (*KVM/WS*). This support was activated in the second scenario (*VMI-PL/VProbes*). The third measurement was run with process life cycle monitoring (*LC*) enabled. This monitoring was performed using the VMI-PL script shown in Listing 3. Finally the last experiment was run with process execution monitoring (*Exec*) enabled. The VMI-PL script for this last case is shown in Listing 4. Due to the fact that VProbes do not support monitoring system calls directly, we used dynamic probes to intercept the execution of the Linux system call dispatcher. It should also be noted that VProbes' `logstr` and `logint` functions were used to log the retrieved data.

Our benchmark results for VMI-PL, as well as for VProbes, are shown in Figure 2 and 3. These figures show the relative change in index value, calculated by UnixBench (each test produces an index value, with higher values meaning higher performance), for each test of the scenario without VMI support. Based on these results, one can see that the performance impact of our VMI-PL implementation, as well as that of VMware VProbes is negligible if no monitoring is active. It is also clear that the performance drops when monitoring is used. As expected, the performance of the virtual machine in tests, such as *Context Switching*, *Process Creation*, and *Shell Scripts* is lower on both systems, if process life cycle monitoring is

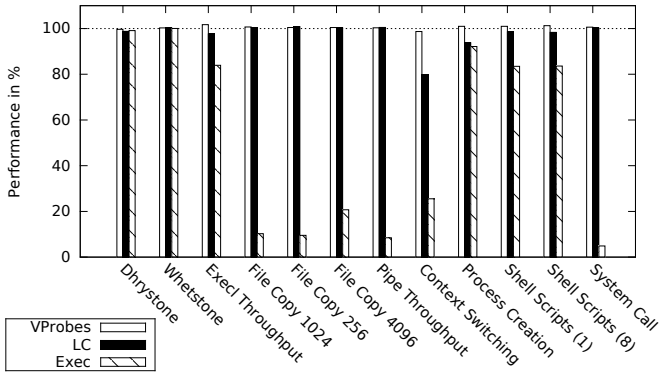

Figure 3: Relative performance impact of VProbes on the UnixBench benchmarks

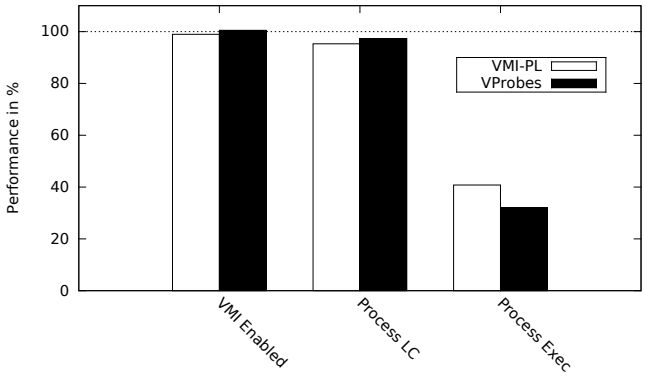

Figure 4: Comparison between VMI-PL and VMware VProbes

active. This is because those tests cause a high number of writes to the `CR3` register, due to the large number of context switches and process creations. The reason for the resulting performance loss is the continued intercept and collection of data at each of these writes. This performance loss is even more severe when process execution monitoring is used. When comparing the worst VMI-PL test results for each of these two cases, we see that the performance loss, caused by process life cycle monitoring, is around 21.3% in the *Context Switching* test. The worst test result, caused by process execution monitoring, in the *System Call* test, is 92.2% lower than the result without any monitoring in place. This decrease in performance comes from the fact that our monitoring intercepts every system call issued during these tests, which naturally slows execution. Since the *System Call* test measures the system's performance with regard to system calls, the highest impact of our monitoring can be seen here.

Figure 4, illustrates the performance of our prototype compared directly with VMware VProbes. This figure is based on the overall system index value, calculated by UnixBench with respect to all performed tests. The percentages show the deviations from the respective system index value, when no monitoring was enabled. The comparison shows that even if our prototype's performance is slightly worse when monitoring support is enabled, as well as when process life cycle monitoring is performed, it is significantly better when monitoring the system calls issued. Due to the fact that VProbes is closed source, we cannot be completely sure about the cause of this performance difference. Nevertheless, from the measurement results, it appears that the use of VProbes' dynamic probe has a higher performance impact than our strategy for monitoring system calls.

Even though the performance impact of the applied monitoring appears to be rather severe, based on the conducted benchmark, it should be noted that this benchmark strives to evaluate the complete capacity of a system. Since real applications rarely make use of a system's capacity to this extent, the performance loss in a real scenario is expected to be lower. In order to evaluate this, we measured the performance impact of our monitoring on the build time of the *Apache* http server, and the request rate handled by this server. In both cases we used Apache version 2.4.4. Due to a lack of space, we present only part of our results here. The complete listing of our results is available in [25]. The monitoring with the highest performance impact in both cases was the process execution monitoring. This monitoring slowed down the build time for the Apache http server by 6.7%, in the case of VMI-PL, and 12.9%, in the case of VProbes. The amount of handled requests per second, which was measured from the host system for a total of 100000 requests, dropped by 18.2%, in case of VMI-PL, and 57.4%, in case of VProbes. Thus, it appears that using VMI-PL for monitoring a virtual machine does not have a too severe performance impact, and furthermore our performance seems better than VProbe's.

8. Conclusion

In this paper, we presented VMI-PL, a monitoring language for virtual machines. This language provides a forensic investigator with a simple tool with which to specify which information should be obtained from inside a guest operating system using VMI. Our approach does not require the user to manipulate or even know the details about the hypervisor. Additionally, the presented language makes it possible to modify the guest's internal state, if necessary. This enables the user to implement more advanced VMI techniques. In contrast to the similar approaches taken by libVMI and VProbes, VMI-PL provides not only access to hardware level data and events, such as memory content or writes to a specific register, but also to operating system level information and events such as information about currently running processes and other OS events such as the start or termination of a process. (Though similar functionality has been implemented by software using libVMI). Furthermore, VMI-PL provides the possibility to monitor data streams to and from the virtual machine, something the other approaches do not.

This can be used, for instance to monitor the virtual machine's network traffic or user input/output.

In addition to the specification of the monitoring language, we also presented an implementation of this language in KVM, evaluated its performance, and compared it to VMware VProbes. This performance evaluation showed that the monitoring can have a high impact on the virtual machine's overall performance, depending on what is monitored. Nevertheless, the comparison with VProbes show that the performance decrease imposed by VMI-PL is similar to, or in many cases lower than, the one imposed by VProbes. Further evaluation on real applications showed that the performance impact in a realistic case is under 20% and therefore probably acceptable in an investigation scenario. Even though the presented prototype implementation of VMI-PL targets KVM, it is also possible to implement support for this language in other hypervisors, which make use of x86 hardware-assistance for virtualization. The KVM prototype implementation is freely available under a liberal software license.

Last, another result of our work is a new classification scheme for VMI techniques. This classification scheme proved to be useful for the specification of our monitoring language, due to the fact that it classifies VMI techniques based on their underlying mechanism of operation.

9. Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments, and in particular Brendan Dolan-Gavitt, for his many helpful comments and suggestions in preparing this paper and improving VMI-PL.

[1] J. C. Flores Cruz, T. Atkison, Digital forensics on a virtual machine, in: Proceedings of the 49th Annual Southeast Regional Conference, ACM, 2011, pp. 326–327.

[2] S. Biggs, S. Vidalis, Cloud computing: The impact on digital forensic investigations, in: Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for, IEEE, 2009, pp. 1–6.

[3] J. Dykstra, A. T. Sherman, Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques, Digital Investigation 9 (2012) S90–S98.

[4] F. Adelstein, Live forensics: diagnosing your system without killing it first, Communications of the ACM 49 (2) (2006) 63–66.

[5] J. C. F. Cruz, T. Atkison, Evolution of traditional digital forensics in virtualization, in: Proceedings of the 50th Annual Southeast Regional Conference, ACM, 2012, pp. 18–23.

[6] A. Fattori, R. Paleari, L. Martignoni, M. Monga, Dynamic and transparent analysis of commodity production systems, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, ACM, 2010, pp. 417–426.

[7] Y. Fu, Z. Lin, Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection, in: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, 2012, pp. 586–600.

[8] VMware, Inc., VProbes Programming Reference (2011).

[9] B. D. Payne, M. de Carbone, W. Lee, Secure and flexible monitoring of virtual machines, in: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, IEEE, 2007, pp. 385–397.

[10] K. Adams, Presenting VProbes, `http://x86vmm.blogspot.se/2007/09/presenting-vprobes.html` (2007).

[11] A. Dehnert, Intrusion detection using vprobes, Tech. rep., VMware, Inc. (2012).

[12] T. Garfinkel, M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: In Proc. Network and Distributed Systems Security Symposium, 2003, pp. 191–206.

[13] J. Pfoh, C. Schneider, C. Eckert, A formal model for virtual machine introspection, in: Proceedings of the 1st ACM workshop on Virtual machine security, ACM, 2009, pp. 1–10.

[14] P. M. Chen, B. D. Noble, When virtual is better than real [operating system relocation to virtual machines], in: Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on, IEEE, 2001, pp. 133–138.

[15] B. Dolan-Gavitt, B. Payne, W. Lee, Leveraging forensic tools for virtual machine introspection, Technical Report GT-CS-11-05, Georgia Institute of Technology (2011).

[16] B. Hay, K. Nance, Forensics examination of volatile system data using virtual introspection, ACM SIGOPS Operating Systems Review 42 (3) (2008) 74–82.

[17] C. Hargreaves, H. Chivers, Recovery of encryption keys from memory using a linear scan, in: Availability, Reliability and Security, 2008. ARES 08. Third International Conference on, IEEE, 2008, pp. 1369–1376.

[18] G. J. Popek, R. P. Goldberg, Formal requirements for virtualizable third generation architectures, Communications of the ACM 17 (7) (1974) 412–421.

[19] S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Antfarm: Tracking processes in a virtual machine environment., in: USENIX Annual Technical Conference, General Track, 2006, pp. 1–14.

[20] J. Pfoh, C. Schneider, C. Eckert, Exploiting the x86 architecture to derive virtual machine state information, in: Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on, IEEE, 2010, pp. 166–175.

[21] S. Krishnan, K. Z. Snow, F. Monrose, Trail of bytes: efficient support for forensic analysis, in: Proceedings of the 17th ACM conference on Computer and communications security, ACM, 2010, pp. 50–60.

[22] L. Litty, H. A. Lagar-Cavilla, D. Lie, Hypervisor support for identifying covertly executing binaries., in: USENIX Security Symposium, 2008, pp. 243–258.

[23] A. Srivastava, J. Giffin, Tamper-resistant, application-aware blocking of malicious network connections, in: Recent Advances in Intrusion Detection, Springer, 2008, pp. 39–58.

[24] S. T. King, P. M. Chen, Subvirt: Implementing malware with virtual machines, in: Security and Privacy, 2006 IEEE Symposium on, IEEE, 2006, pp. 14–pp.

[25] F. Westphal, Vmi-pl: A monitoring language for infrastructure-as-a-service platforms, Master's thesis, Hasso-Plattner-Institute, University of Potsdam (2014).

[26] A. Dinaburg, P. Royal, M. Sharif, W. Lee, Ether: Malware analysis via hardware virtualization extensions, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, ACM, New York, NY, USA, 2008, pp. 51–62.