Assessing performance overhead of Virtual Machine Introspection and its suitability for malware analysis

Master's Thesis in Technology
University of Turku
Department of Future Technologies
Networked Systems Security
2020
Sebastian Paakkola

UNIVERSITY OF TURKU
Department of Information Technology

SEBASTIAN PAAKKOLA: Assessing performance overhead of Virtual Machine Introspection and its suitability for malware analysis

Master's Thesis in Technology, 105 p., 0 app. p.
Networked Systems Security
June 2020

Virtual Machine Introspection is the process of introspecting guest VM's memory and reconstructing the state of the guest operating system. Due to its isolation, stealth and full visibility of the monitored target, VMI lends itself well for security monitoring and malware analysis. The topics covered in this thesis include operating system and hypervisor concepts, the semantic gap issue, VMI techniques and implementations, applying VMI for malware analysis, and analysis of the performance overhead.
The behaviour and magnitude of the performance overhead associated with doing virtual machine introspection is analysed with five different empirical test cases. The intention of the tests is to estimate the costs of a single trapped event, determine the feasibility of various monitoring sensors from usability and stealth perspective, and analyse the behaviour of performance overhead.
Various VMI-based tools were considered for the measurement, but DRAKVUF was chosen as it is the most advanced tool available. The test cases go as follows. The chosen load is first executed without any monitoring to determine the baseline execution time. Then a DRAKVUF monitoring plugin is turned on and the load is executed again. After both measurements have been made, the difference between the two execution times is the time spent executing monitoring code. The execution overhead is then determined by calculating the difference between the two execution times and dividing it by the baseline execution time.
The disc consumption and execution overhead of a sensor, which captures removed files is small enough to be deployed as a monitoring solution. The performance overhead of system call monitoring sensor is dependant on the number of issued system calls. Loads which issue large numbers of system calls cause high performance overhead. The performance overhead of such loads can be limited by monitoring a subset of all system calls.

Keywords: Virtual Machine Introspection, performance overhead, malware analysis

Contents

1 Introduction **1**

2 Operating system concepts **3**

2.1 Process 3

2.2 Windows process and threads 5

2.3 Linux process and threads 8

2.4 System calls 9

2.5 Memory management 10

3 Hypervisor and virtualization techniques **17**

3.1 Benefits and opportunities of virtualization 17

3.2 Virtulization techniques 19

3.3 Virtualizing the CPU 20

3.4 Virtualizing the memory 22

4 Virtual Machine Introspection techniques **24**

4.1 What is VMI? 24

4.2 Bridging the semantic gap 26

4.2.1 In-VM 27

4.2.2 Out-of-VM delivered 27

4.2.3 Out-of-VM derived 29

4.3 Trapping system calls 32
4.4 Paging system 34

5 Implementations of VMI 37

5.1 Livewire 37
5.1.1 VMI-based Intrusion Detection System 37
5.1.2 Design of Livewire 38
5.2 Ether 40
5.2.1 A Transparent External Malware analyzer 40
5.2.2 Implementation of Ether 41
5.3 Nitro 44
5.3.1 Hardware-based system-call tracking mechanism 44
5.3.2 Implementation of Nitro 45
5.3.3 Future direction 46
5.4 DARKVUF 47
5.4.1 Implementation 47
5.4.2 Usage 51

6 LibVMI: Virtual Machine Introspection library 54

6.1 General capabilities 54
6.2 LibVMI usage 57
6.2.1 Initialization 57
6.2.2 LibVMI API overview 59

7 Malware analysis with virtual machine introspection 64

7.1 Behavioural analysis 64
7.2 Code analysis 66
7.3 Memory analysis 67
7.4 DRAKVUF as a malware analysis platform 68

8 Estimating the performance overhead of Virtual Machine Introspection 71

8.1 Detecting the monitoring solution from the VM 71

8.2 Performance degradation caused by Virtual Machine Introspection 73

8.3 Motivation and research goals 74

8.4 Test scenarios 76

8.4.1 Test 1: Calculating Fast Fourier Transforms 77

8.4.2 Test 2: Novabench desktop benchmarking tool 77

8.4.3 Test 3: System call monitoring with simulated browser usage . . . 78

8.4.4 Test 4: File carving overhead measurement 80

8.4.5 Test 5: Pool allocation monitoring overhead 82

8.5 Results 83

8.5.1 Test 1 results: FFT overhead 83

8.5.2 Test 2 results: Novabench scores analyzed 84

8.5.3 Test 3 results: System call monitoring 85

8.5.4 Test 4 results: File carving overhead 92

8.5.5 Test 5 results: Pool allocation monitoring overhead 95

8.5.6 Summarizing results 95

9 Conclusions 98

References 102

1 Introduction

Malicious programs are becoming increasingly advanced and aware of their operating environment. It is common for them to include modules that attempt to detect and hinder the use malware analysis environments and tools such as debuggers [1]. To manage the growing number of malicious programs, cybersecurity community needs tools to reliably study and analyse them. As malware can change its behaviour if it observes the monitoring environment, new stealthier methods are required.

By placing the monitoring environment outside the virtual machine, one gains stealth, tamper resistance and visibility [2]. However, by doing so one also looses the operating system specific semantic knowledge about the meaning of the observed bits and bytes [3]. Reapplying the semantic knowledge is one of the key challenges in virtual machine introspection and it is known in literature as *bridging the semantic gap*.

When virtual machine introspection is applied in practise, another challenge is gaining information on interesting behaviour inside the monitored environment. VMI-based solutions have creatively leveraged hardware functionality to transfer the control to the hypervisor when events such as system calls are issued [4][2].

When execution is transferred to the hypervisor level, some performance overhead is introduced. Managing the magnitude of the performance overhead is an important design goal for VMI-based solutions. Often times performance overhead management requires balancing between additional information on the activities of the guest and high performance overhead. The more events you trap, the more performance overhead is included.

The following research questions were formulated for this thesis.

RQ1 What are the benefits of Virtual Machine Introspection when compared to in-VM techniques in the context of malware analysis?

RQ2 What are the VMI-based malware analysis tools available and how do their properties differ?

RQ3 How can Virtual Machine Introspection be applied to malware analysis?

RQ4 What is the magnitude and behaviour of performance overhead associated with running VMI-based malware analysis tools?

RQ4 is the main focus of this thesis. It analyses the general applicability of VMI from usability and stealth perspective. The usability perspective includes analysing if VMI-based monitoring solutions can be deployed to production environments. The stealth perspective means evaluating whether the monitoring platform can be detected from within the analysis environment.

The remainder of this thesis is structured as follows. Chapters 2 and 3 cover the operating system and virtualization related topics required to understand virtual machine introspection. Chapter 4 introduces virtual machine introspection as well as the semantic gap issue. Chapters 5 and 6 deal with practical VMI-based solutions and libraries. Chapter 7 covers malware analysis in the context of virtual machine introspection and Chapter 8 analyses the performance overhead of virtual machine introspection.

2 Operating system concepts

2.1 Process

Modern operating systems allow the user to simultaneously run multiple programs on the same machine. To enable this parallelism, the concept of process was created. Each process is an instance of a running program with its own separate address space, thread of execution and state.

A process can have three different states. It is either currently in execution (running), waiting for CPU time (ready) or waiting for some external event such as an I/O to happen (blocked). The state of a process change from running to blocked, once it reaches a point in execution where the program performs a blocking system call. For example, in UNIX, when a process reads from a pipe and there is no input yet available, the process enters blocked state. Once the input becomes available, the state changes to ready. Once the operating system decides that a ready process can start running again, the state switches back to running. The state can also change from running to ready if the operating system chooses that it is time to let the other processes have some CPU time. [5]

To allow processes to have different states, one needs to be able to stop and resume the processes in a structured manner. Once a process is stopped, the information state of the process needs to be saved somewhere so that when the process is later resumed, the execution starts exactly from the same spot. This state information is saved to the process table. [5]

The process table contains a single entry for each process currently present in the machine. It holds information about process management, memory management and file management. Precisely which items are present in the process table vary largely between operating systems. In Windows environments, the user can read some of the information present in the process table from the task manager. This information includes the process name, ID, state, user, amount of allocated memory and CPU time. In Linux environments, the users can view this information with the command ps. In addition to this management data, the process table entry contains the pointers to open files, program counter values and registers. [5]

Each process can have multiple threads inside them. These threads share the same address space, open files and global variables, as those items are per-process, but they have their own program counters, registers, stack and state. The states that a thread can have are the same as the states of a process and the transitions between those states are also the same. [5]

Threads allow the program to perform multiple separate tasks at the same time. For example, a word processor could have threads for formatting the document, taking used input and performing periodical saves. As these tasks are separate, each thread can independently perform its duties and thus increase the performance of the program. [5]

Threads introduce some extra complexity as each process no longer has a single thread of execution (and a single program counter and stack), the operating system needs some way to handle it. Depending whether threads are implemented in the user space or in the kernel, the solution varies a bit.

If the threads are implemented in the user space, a run-time system is needed to handle the threads and each process has its own thread table. The run-time system handles transitions and scheduling between threads and the kernel is not concerned about the user-level threads. Implementing threads in the user space is faster as it can use procedure calls rather than trapping and flushing context. On the other hand, if threads are implemented in the user space, blocking system calls become an issue. If the thread would call a blocking system call, the entire process would enter blocked state as the kernel is not aware of threads and the execution would switch to some other process even if there were threads in ready state. To overcome this issue, one would need to either change all blocking system calls to non-blocking ones or write wrapper code to check whether a system call would block. [5]

Implementing threads in the kernel requires no run-time system or per process thread tables. Instead the kernel maintains a single thread table, which keeps track of all threads within a system. The thread table holds information to necessary to resume, stop and switch the thread currently in processing such as program counter values and registers. Implementing the threads in kernel would remove the issue of blocking system calls as if a thread would issue a blocking system call, the kernel could simply switch to another thread of that process or decide to switch to another process. [5]

2.2 Windows process and threads

In Windows environments, there are multiple different data structures that are related to processes. Each process is represented in the kernel by a structure called executive process block (EPROCESS). This structure contains attributes and pointers to other data structures that are related to that process. For example, this structure contains pointers to the kernel's data structure executive thread block (ETHREAD), which is a data structure that represents threads in the Windows kernel. [6]

The Windows operating system structures can be viewed with WinDbg tool. For example, command dt _EPROCESS shows the structure of the EPROCESS. By adding a recursive switch -r1, the debugger also shows the structures one recursion level deep. For example, in the case of EPROCESS, it would also show the structure of KTHREAD. As the EPROCESS structure contains so many attributes and pointers that covering them all

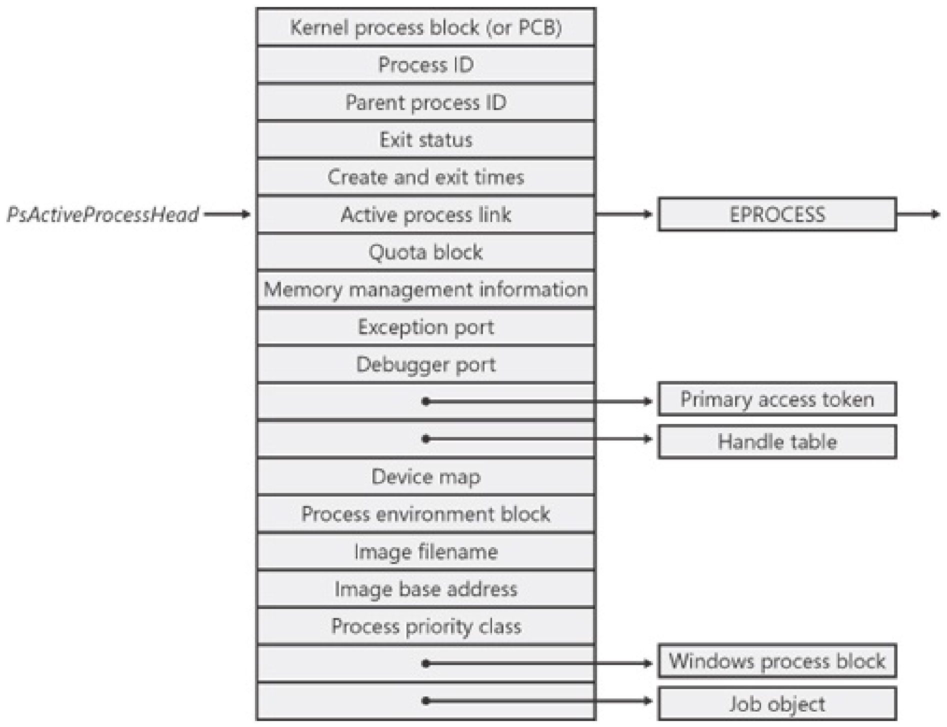


Figure 2.1: Key fields of EPROCESS structure. Image from [6]

would not be meaningful for the purpose of this thesis, I will focus only on a subset of the attributes and structures. [6]

Many of the attributes in the EPROCESS structure, such as process identification, are self-explanatory. Others, particularly the ones that point out to other structures, might need more explanation. The first attribute in the EPROCESS structure is another structure called KPROCESS (or the process control block). It contains scheduling related information such as total kernel and user time and list of kernel threads belonging to the process. It also has the base to the process directory table. The quota block structure in EPROCESS describes the limits that are set to the process. The quota block can be shared by multiple processes. The structure contains pointer to the image file, and the name if the image file. [6]

The EPROCESS contains links to the next (flink) and previous (blink) EPROCESS blocks. The kernel variable PsActiveProcessHead points out to the first item in the list. Volatility's pslist plugin uses them when listing out currently running processes [7]. As the plugin works by walking the double linked list, it cannot detect hidden or unlinked processes.

The EPROCESS block and the related data structures exist in the system address space, but it also contains a pointer to Process Environment Block, which is in process address space. The PEB contains information that needs to be accessible to user-mode code. It is used by the image loader, heap manager and other system DLLs. The Process environment block contains the image base address, list of loaded modules, information about process heaps and process parameters structure (RTL_USER_PROCESS_PARAMETERS), which contains for example the exact command line string that was used to launch the process. [6]

In Windows environments, threads are the kernel level of abstraction for scheduling. Each process can have multiple threads, that can run simultaneously. ETHREAD is the kernel's data structure for a thread. Similarly to EPROCESS, it is a complicated data structure with multiple different attributes and structures. It too resides in the system address space as do the structures it points out to apart from the thread environment block (TEB). [6]

ETHREAD contains a KTHREAD structure (also called thread control block), which like its process counterpart, contains scheduling information such as execution time and priority information. KTHREAD contains thread's identification information, pointer to its owning process, pointer to the access token, which specifies the security information and a list of pending I/O requests. [6]

Thread environment block stores context information for the image loader and different DLL's. The TEB contains pointers to the owning process environment block, identification information as well as stack information (stack base and stack limit). It also has a pointer to its local storage. [6]

2.3 Linux process and threads

In Linux environments all currently active processes are represented by a doubly linked list of task_struct data structures called the task vector. The task_struct data structure is the kernel's representation of processes and it is defined in the linux/sched.h include file in the kernel source-code. For each process the task_struct is always kept in memory. [5][8]

In Windows environments threads are represented by the ETHREAD structure. Linux represents all execution contexts, threads and processes alike, with the task_struct data structure. A single-threaded process would have only one task structure whereas a multi-threaded process would have one per thread. [5]

Similarly to its counterpart EPROCESS in Windows environments, the task_struct structure contains multiple fields some of which are pointers to other data structures such as open files and signal handles. It too is a large and complex data structure. Having pointers to other data structures instead of containing all information on the task_struct directly allows to page out information that is not currently needed and save memory space. For example, information about file descriptors may be paged out when the process is not in memory. [5]

A Linux process can have four different states which are running, waiting, stopped and zombie. The state running indicates that a process is either currently running in the CPU or that it is ready to be run. The state waiting indicates that the process is waiting for an event to happen or for a resource to become available. The waiting state is different types interruptible and uninterruptible. The state stopped occurs for example, when a process is stopped for debugging purposes. The state zombie indicates that the process is terminated but it still has a task_struct data structure in the task vector. The reason for the task_struct remaining in the task vector is that the parent process has not made a wait() system call yet. Before the system call is issued, the child process cannot discard data in its descriptor (task_struct). [8]

The task_structure contains various identifiers for the process. Each process has an unique process identifier, task identifier, group and user identifiers. Those identifiers are used to manage which resources (files and devices) the process has sufficient rights to access. [8]

In Linux systems, each process, apart from the initial process, has a parent process. Whenever a new process is created, a fork() system call is issued. The system call functions by cloning the parent process and then continuing execution from the next instruction. It copies all fields from the parent's descriptor to the child's descriptor (apart from some identifiers). The fork system call takes no parameter and it returns a zero for the newly created child process and the PID of the child process for the parent. The return value of the fork system call is used to differentiate between the parent and the child process. The task_struct data structure maintains links to its parent process and child processes. [5][8]

The data structure also has scheduling related information such as process priority, CPU time consumed and time spent sleeping. The parameters are used by the scheduler to allocate CPU time to runnable processes. [8]

2.4 System calls

A system call is a programmatic way for user level programs to request services from the operating system kernel. The services provided through the system call interface include process and thread management, file access, I/O handing, networking and so on. Whenever a program wishes to use these operating system managed resources, it needs to request them from the operating system by issuing a system call from the operating system's system call interface. The exact implementation of system calls varies between operating systems and even operating system versions. [5]

In Windows environments, system calls are implemented in the Native NT API. Many native NT system calls operate with kernel-mode objects. The kernel mode objects can represent files, pipes, threads and so on. When a system call, which creates a kernel-mode object is issued, the caller program is then given a handle to the object. The handle can then be used to perform subsequent operations on the object (the handle is passed as a system call parameter). Each process has their own handles, which are stored in their handle table. [5]

When a Windows program wishes to use operating system resources, it calls a procedure from the Win32 API. The Win32 API then issues the correct system calls required to perform the requested task. Some Win32 API calls perform their duties entirely in user space and thus do not issue any system calls. [5]

In Linux environments, system calls are issued by calling library procedures. The library procedures are implemented in assembly language. The library procedure puts system call parameters to the correct registers and then issues a trap instruction to change the execution to the kernel mode. The kernel-mode handler then inspects the system call number to determine which system call was issued and proceeds accordingly. The POSIX standard defines the library calls which compliant operating systems need to implement. [5]

To summarize, a system call is a method for user level programs to request operating system services. Whenever a system call is issued, the execution is transferred to the operating system (trapped to kernel), which then performs the requested task. System calls are used to perform tasks such as process management, file management, communication, information maintenance, security, and device management. [5]

2.5 Memory management

All processes reside in physical memory. They have their own individual address space, which they can reference. The address spaces should be separated and protected so that a process can only access its own address space and not the address space of other processes or the operating system. Memory management is one of the key responsibilities of an operating system.

In the simplest scenario, processes reference absolute memory addresses. This, however, creates a situation where it is possible for a process to accidentally or intentionally access and modify the address space of another process or even the operating system. This can lead to unexpected behaviour and can crash the system. It is usually not possible to have multiple processes running on the machine if processes directly reference absolute memory addresses. [5]

The concept of virtual address was created to address this issue. Each process has their own address space (set of memory addresses) that they control. Instead of referencing absolute memory addresses programs reference virtual addresses and the MMU (memory management unit) then converts the virtual addresses into physical memory addresses. There are multiple different ways that can be used to implement this.

One way to achieve virtual addresses is to use base and limit registers. All processes are loaded into consecutive chunks of memory, wherever there is enough room to hold the entire process. Whenever a program makes a memory reference, the CPU automatically adds the base register to the virtual address and checks if the resulting address is valid (not over limit). if for some reason the address violates the limit register (process tries to access memory that is not within its address space), a fault is generated and the CPU aborts. The base register contains the physical address of where the program begins, and the limit has the length of the program. Some variations of base & limit register do not use the limit register at all. [5]

Using base and limit registers requires the entire process to be loaded in the memory. For a new process to enter, there needs to be a hole that is large enough to hold the entire process. All processes that are loaded into memory have their own values for base and limit registers. Whenever a context switch is made (the process that is currently running is switched), also the contents of the base and limit register need to be switched to match the correct program. [5]

In many cases the total amount of RAM is not sufficient to hold all processes at the same time. More dynamic ways to allocate memory are needed. The operating system needs to manage the increased complexity.

A rather straightforward approach to accomplish dynamic memory allocation is memory swapping. The physical memory is divided into allocation units. The size of each allocation unit is usually between few words and several kilobytes. A process takes up some multiple of allocation units which forms the address space of the process. Once the process is switched back out to disk, the allocation units are marked as holes and the contents are written back to disk if needed. The operating system needs to know which allocation units are currently occupied by a process and which are holes. The two main methods to keep track of memory are bitmaps linked lists. [5]

In bitmap, each allocation unit is represented by a single bit. That bit is used to indicate if the allocation unit the bit represents is a hole or if it is occupied by a process. Whenever a new process is loaded, the operating system needs to search through the bitmap to find a hole that is large enough to hold the process. When a hole is found, the operating system loads the process there and changes the bitmap to match the new situation. Similarly, whenever a process is switched back to disk, the operating system needs to update the bitmap. [5]

Whenever a new process is loaded into memory, the operating system needs to search through the entire bitmap for a large enough hole. This search is considered to be slow.

An alternative to bitmaps is linked lists. The list contains information about the segment. Each item in the hole specifies a single hole or a process and contains information such as the type of the segment (hole or process), the starting address of the segment, length of the segment and a pointer to the next item on the list. To speed up list management when a process is removed, it is beneficial to also have a pointer to the previous value (speeds up finding the previous neighbour). [5]

When a process leaves, the operating system needs to modify the list to match the current situation. With linked lists the modification is not as simple as it is with bitmaps. When a process leaves, the space that it currently occupies becomes a hole. The new hole needs to be merged with the neighbouring holes to avoid memory becoming fragmented into small useless pieces. When each segment is linked to the next segment and the previous segment, it is easy to find the neighbours that need to be considered.

In the above discussion, I have assumed that the linked list is a single list containing both processes and holes and is sorted by address. This is not necessarily always the case as the list can also be sorted out by size or we can have separate lists for processes and holes. Having separate lists for processes and holes speeds up memory allocation but slows deallocation as the neighbours need to be found to merge holes.

Having a list of holes that is sorted out by hole size speeds up allocation algorithms such as best fit and worst fit as they no longer need to search through the entire linked list to find the correct hole. [5]

The allocation unit size is an important parameter, especially with bitmaps as each allocation unit is represented by a single bit. With small allocation unit size, the bitmap becomes larger and takes up more space. On the other hand, when the allocation units are smaller, the sizes of processes are closer to multiples of allocation units and there is less slack space. [5]

Virtual Memory

With memory swapping, the entire process is loaded into the memory. However, sometimes the size of the program exceeds the size of memory. Such programs cannot be run with systems that use swapping as their allocation technique. Also, if the entire process needs to be loaded into memory, much fewer processes can be run simultaneously as the combined size of all processes quickly exceeds the total amount of memory. Memory swapping techniques are also relatively slow as disk operation. [5]

Virtual memory can be larger than physical memory. With virtual memory, the memory is divided into fixed size pages. Each page is a continuous range of addresses and can be mapped to physical memory. The element in physical memory that corresponds to virtual page is called a page frame. All pages do not need to be present in memory to run a program. This feature saves up lots of space in memory as the program does not need to be entirely in memory. [5]

Whenever a program makes a memory reference, the virtual page is mapped to physical memory. The virtual address is sent to the MMU and the MMU maps the virtual address to physical address. If the virtual page is mapped to a page frame, the execution can continue normally, and the virtual address is translated to the correct address in physical memory. If the referenced virtual page is not present in memory, a page fault is raised, and the operating system is alerted to fetch the missing page from disk. [5]

Virtual address is split into a virtual page number and offset. The high order bits represent virtual page number and the low order bits represent the offset. When a virtual address is mapped to a physical address, the bits that represent virtual page number are replaced with bits that represent the page frame number. The correct page frame number is taken from page table. [5]

In a manner of speaking, page table is a function that maps virtual page numbers into page frame numbers. The page table can be indexed by virtual page number. For example, a page table entry in spot 12 represents the virtual-to-physical mapping for virtual page 12. Each page table entry contains an indicator that tells whether a page is present in virtual memory or not. This indicator is the present/absent bit. Page table entry also contains the page frame number for that virtual page. The operating system also needs to know if the page frame needs to be written back to disk. The modified bit is used to indicate if the page has been modified and needs to be written back to disk when it is removed from memory. Page table also tells what kind of access is permitted to the page (read, write, execute). [5]

For paging to work efficiently, the virtual-to-physical mapping needs to be fast as it is done on every memory reference. As the page table is normally located in memory and memory look-ups take a long time, simply searching the page table every time is not efficient. Widely used approach to speed up paging is Transition Lookaside Buffer (TLB). TLB is based on the fact that most programs tend to make a large amount of references to a small number of pages. The principal idea of TLB is to keep frequently needed mappings somewhere they can be accessed fast (requires special hardware components). [5]

Now when a program makes a memory reference, it is first checked if the requested page is present in the TLB. All entries in the TLB are searched in parallel so searching the TLB is very fast. If the requested page is not in TLB the system does a regular page table lookup (TLB miss). Once it finds the correct page frame from page table, it replaces one entry from the TLB and moves on. Next time a reference is made to the same virtual page, the mapping is already present in TLB and the address can be mapped fast. Whenever a mapping is evicted from TLB, the modified bit needs to be written back to memory. [5]

TLB is often done with hardware, but it is also possible to implement it in software. When TLB is implemented with software, the entries in it are explicitly loaded by the operating system. Now if a TLB miss occurs, a TLB fault is generated and the execution is transferred to the operating system. The operating system then finds the correct page and replaces an entry from the TLB. Once the TLB entry is replaced, the execution is returned to the program. [5]

TLB misses are much more frequent than page faults. With large enough buffers implementing TLB in software becomes efficient enough. This in turn frees up space from the CPU chip to other features as the MMU is now much simpler and requires less space. [5]

With large address spaces, page tables will be large. Multilevel and inverted page tables have been developed to reduce the memory space the page table needs.

The idea behind multilevel page tables is to not keep all page tables in memory at all times. Page tables that are not needed are not kept in memory either. For example, to create a two-level page table with 32-bit addresses the first 10 bits of a virtual address could be used to indicate which entry is selected in the top-level page table, the following 10 bits are used to select the correct page in the second-level page table and the remaining 12 bits are the offset. The top-level page table covers the entire virtual memory and the entries in it represent larger blocks of virtual memory. There is one second-level page table per top-level page table entry. A multilevel page table is beneficial as programs often only use the top and the bottom parts of their virtual address space. Instead of keeping the entire page table in memory, we keep only the top-level page table with a few second level page tables. The multilevel page table can be deeper than only two levels. [5]

Inverted page table is another attempt to combat the size of page tables with large virtual address space. Instead of mapping every virtual page to a page frame, only those pages that are currently present in memory are mapped. The inverted page table keeps track of which process and which virtual page is located in the page frame. With inverted page tables, we save a lot of space but the virtual-to-physical mapping becomes a lot harder. TLB can be used to reduce the cost of a page table lookup. [5]

3 Hypervisor and virtualization techniques

3.1 Benefits and opportunities of virtualization

Virtual machines and virtualization have lately become popular due to enterprises moving large portions of their IT infrastructures to the cloud. Virtualization allows companies to host multiple different servers on the same physical machine. Each virtual machine running on the same host is separated from each other. This means that even if one of those virtual machines should crash, the others would remain unaffected by it. Each virtual machine appears to be a complete machine with its own configurations, services and software. [5]

Virtualization has brought multiple new business models. There are many different as-a-service models that essentially sell virtualized equipment at different abstraction levels. Infrastructure-as-a-service models sell virtualized computing resources over the internet. Various monitoring and logging services can be included in the contract that take some of the management responsibilities away from the client. Platform-as-a-service builds on top of IaaS by also providing the client the operating system or databases. With PaaS the client needs not to worry about maintaining the operating systems and infrastructure as they are contracted to the cloud service provider. The client can install and run any applications or services that the platform supports. Software-as-a-service provides the client with readily installed application. The client simply logs in to the service. One familiar example of SaaS is Google Docs.

According to Tieto State of cloud in the Nordics Cloud Maturity Index 2019 [9] report roughly 22 % of all IT expenditures in the Nordics are cloud expenditures. The report predicts that the overall portion of cloud expenditures will rise to roughly one-third by 2022. They justify this prediction by stating that different as-a-service models will be used more widely and in more critical areas than before. In fact the report states that over 80 % of interviewed IT-decision makers estimated that their use of cloud services would increase.

Virtualization allows the organizations to dedicate machines to host only a single service without increasing the costs requiring them to have physical machines per hosted service. Separating the services to different machines is a desirable goal because it increases the robustness and security of the system. Should one service become compromised, it does not immediately lead to other services being compromised too. On the other hand, should a hardware component fail, all virtual machines running on top of it would also break.

Virtualization also increases computing resource utilization as the slacking resources can be assigned to another virtual machine that needs them. In fact, it is possible to dynamically allocate resources to virtual machines and containers. Being able to dynamically allocate resources allows the organizations to easily increase the computing capabilities during peak hours or if the demand starts to rise.

Virtualization and different as-a-service models also make it easier for people to access their information, files or applications. Instead of requiring the user to install the application on their own device or store files locally, the user can use any device with internet connection to access the files or application.

3.2 Virtulization techniques

In this thesis, we are only interested in full virtualization. In full virtualization, the goal is to present a virtual machine that acts like the underlaying hardware. The virtualization effort is focused on virtualizing the physical hardware components such as the CPU and memory. On top of those virtualized hardware, one can install any operating system that is supported by the hardware. The operating system is not aware that it is running on virtual machine and does not require special modifications to it. [10]

Full virtualization is not the only type of virtualization. One commonly used virtualization method is *OS-level virtualization*. The goal is not to virtualize the hardware set but to have multiple isolated instances of user space. The operating system kernel is shared by the virtualized user spaces. These instances are called containers. They contain the code and dependencies necessary to run the application that is being containerized. [11]

Containers share similar security benefits as virtual machines as the applications are isolated in their own containers. Computing resources can be allocated per container. Containers are lighter than virtual machines as containers do not require an operating system per container. They also take less space as container images are typically tens of megabytes in size while virtual machines, having a complete operating system, usually require several gigabytes. Examples of container management software are Docker Platform and Google Kubernetes. [11]

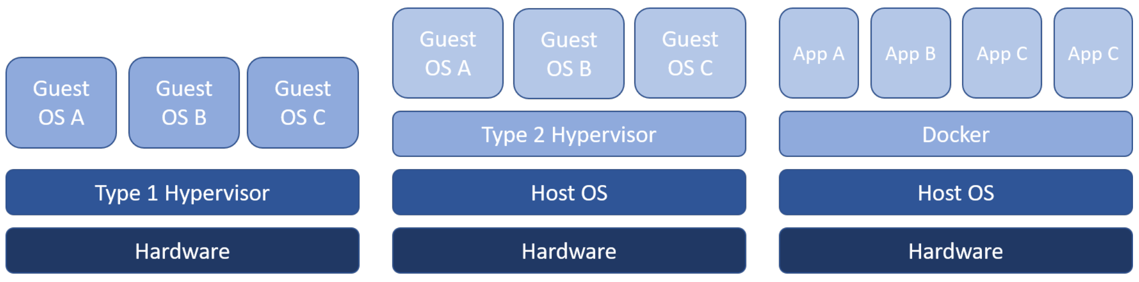


Figure 3.1: Full virtualization with type 1 and 2 hypervisors and OS-level virtualization with Docker.

Paravirtualization is a virtualization technique, where the guest is aware that is being virtualized. The hypervisor offers a software interface that directly tells that the environment is virtualized. This interface offers the guest various hypercalls to do sensate instructions such as page table modifications. When the sensitive operations are performed in cooperation with the hypervisor, the resulting system can be faster. However, paravirtualization requires that the guest operating system is modified to be able to cooperate with the hypervisor. [10]

3.3 Virtualizing the CPU

Hypervisor is the element that is tasked with creating the illusion of multiple different devices. Essentially the hypervisor creates multiple virtual copies of the under-laying hardware and presents those copies to the virtual machines. In other words, the hypervisor has direct control over the physical hardware instead of the operating system. In some literature hypervisors are also referred to as virtual machine monitors (VMMs). [5][10]

There are three dimensions where hypervisors should perform well. Firstly, the hypervisor has full control over the virtualized resources. All traps and interrupts originating from the guest operating systems go to the hypervisor, which will then emulate the behaviour of the actual hardware (interposition). Secondly, all virtual machines are isolated from each other. A virtual machine is not able to see or interact with any process running on another virtual machine on the same physical hardware. Similarly, the virtual machines are isolated from the host operating system. Thirdly, the hypervisor has access to all states of the virtual machines as it is the only element with full access to the physical hardware. [10]

In their paper, Popek and Goldberg introduced the criteria for a machine to be virtualizable – a machine is virtualizable only if the set of sensitive instructions is a subset of privileged instructions. A sensitive instruction is an instruction that operates differently when executed in kernel mode than when executed in user mode. More specifically "an instruction is control sensitive if it attempts to change the amount of (memory) resources available or affects the processor mode without going through memory trap sequence. [12]" An instruction is privileged if it gets trapped when executed in user mode but does not get trapped when executed in kernel mode. The virtualization criterion essentially means that all sensitive instructions get trapped when they are executed in user mode. [12][5]

Any program running on the virtual machine should behave identically when compared to the same program running on bare hardware. Before 2005 and the introduction of hardware level support for virtualization from Intel (Intel Virtualization Technology) and AMD (Secure Virtual Machine), this was not the case as the virtualization criterion was not satisfied. [10][5]

Even though the hardware did not directly support virtualization, it was possible to have virtual machines before 2005. Instead of using trap-and-emulate schema, the hypervisor attempted to get rid of all instructions that were sensitive but not privileged (instructions that violated the virtualization criterion). This was done by using the four protection rings in the x86 processor architecture and binary translation. [10]

The x86 architecture has four protection rings. The kernel operates on ring 0 and the user applications operate on ring 3. Rings 1 and 2 are not usually used. In a virtualized environment, the hypervisor is the only one that runs in ring 0 as it is the most privileged level. The kernel of the guest operating system is moved to protection ring 1. Therefore, the hypervisor is in a more privileged position compared to the guest operating system kernel. [10]

The hypervisor modifies the kernel code of the guest operating system prior to execution to replace all sensitive instructions with hypervisor procedure calls that handles them. The translation is done one basic block at a time (sequence of instructions that ends with a branch) and the translated sequence is cached for later use. The translation is often done even on code that could be made to trap (instructions that are both privileged and sensitive). The reason for doing it is that doing binary translation leads to better performance as trapping is expensive. [5]

A virtual machine runs as a user process in user mode. As it is in user mode, it cannot run sensitive instructions without a trap. Still, the guest operating system thinks it runs in kernel mode even though it is really run in user mode. This mode is called the virtual kernel mode. With hardware that supports virtualization, whenever the guest operating system runs instructions that are only allowed in kernel mode, a trap to the hypervisor occurs. The hypervisor must then check if the instruction came from the guest operating system or from a user program within the guest operating system. If the instruction came from the guest OS, the hypervisor can let the CPU carry out the instruction. If the instruction came from a user program, the hypervisor must emulate what the hardware would do if the same situation happened without virtualization. [5]

There are two different types of hypervisors – type 1 and type 2. The difference between the two is that type 1 hypervisors operate directly over the hardware whereas type 2 hypervisors operate on top of a host operating system. Type 2 hypervisors rely on the host operating system to allocate and schedule resources. Xen, vSphere and Hyper-V are examples of type 1 hypervisors. Products such as VMware and KVM are type 2 hypervisors. [5]

3.4 Virtualizing the memory

Nearly all modern operating systems support virtual memory. Virtual memory is essentially a mapping from virtual address pages to physical page frames. That mapping is defined by page tables which are kept in memory. The location of the top-level page table is set to a control register (CR3) by the operating system. [5]

Virtualization makes memory management more complicated as all virtual machines need to keep their own page tables. Problems arise when two different virtual machines try to map their virtual pages to the same physical page frame. To solve this problem, the hypervisors needs to keep a shadow page table for each hosted virtual machine. The shadow page table maps the virtual machines virtual pages to the page frame the hypervisor has given it. [5]

This solution does have its flaws. Whenever the guest operating system decides to change its page table, the shadow page table also needs to be updated. However, the guest operating system can change its own page table without having to trap to the hypervisor, so the hypervisor does not become aware of the change. [5]

There are two approaches to solving this issue. The hypervisor can set the virtual machines page table as read-only and now if the VM tries to modify its page tables a fault occurs, and execution is transferred to the hypervisor. The hypervisor then needs to analyse the instruction to determine what the guest operating system wanted to do and update the shadow table. The second alternative is to let the guest OS change its page table freely. When the VM tries to access new pages, a fault occurs and the hypervisor gains control. The hypervisor then needs to check the guest OS's page tables to see if there are any new mappings it should know and then modify the shadow table. Both of these techniques are expensive as they lead to a VM exit (transferring control to the hypervisor). [5]

As the cost of handling shadow tables is high, the hardware now supports nested page tables. Nested page tables aim at reducing the overhead by handling page table manipulation in hardware without traps. Nested page tables translate the guest virtual address to guest physical address and then use the nested page table to find the host physical address. Nested page table look-ups need to be done on every level of the multilevel page table hierarchy. [5]

4 Virtual Machine Introspection techniques

4.1 What is VMI?

Virtual Machine Introspection is the process of monitoring live virtual machines from the hypervisor to gain insight on the activities within the guest operating system. This monitoring approach has gained popularity and is the focus of many security solutions and platforms because the hypervisor, the privileges of which the security solution inherits, has full visibility and control over the monitored guest OS. Placing the monitoring solution outside the monitored VM is desirable because sophisticated malware is often capable to circumvent and tamper with in-VM security solutions. As the VMI security solution resides outside the VM, it is highly resistant to tampering (hypervisor's isolation feature). [3]

As the hypervisor has full control over the virtualized set of hardware, it can inspect and modify any guest OS artefact (CPU registers, memory), which leads to high evasion resistance. Due to the hypervisor's interposition feature, all traps and interrupts are handled by the hypervisor. This enables the VMI security solution developer to create their own handlers and modify the return values from these interrupts. To increase their capabilities, VMI solutions such as Ether and Nitro force traps on system calls that would not natively cause VMEXITs. Increasing the amount of VMEXITs does, however, leads to higher performance overhead. [3]

As we are working with full virtualization, the guest operating system is not aware that it is running in a virtualized environment. Sophisticated malware often looks for signs that it is running in a virtualized environment and can modify its behaviour if it thinks it is being monitored. Signs that indicate that the environment is virtualized are some virtualization specific registry keys, in-VM tools such as VMware tools or timing differences. As malware's behaviour can change, it is important to be able to monitor it undetected. VMI solutions do not need any in-VM tools or agents to function so they are less likely to expose their presence. [13]

The downside of placing the monitoring solution outside the monitored host and into the hypervisor is that it lacks all high-level semantic information and does not have access to the APIs in the guest OS. The hypervisor's view of the guest operating system is just raw bits and bytes as the primary job of a hypervisor is to create multiple copies of the underlaying physical hardware (full virtualization). The guest operating systems are then installed on top of the virtualized hardware. Thus, the guest operating system operates on a higher abstraction level. The only way for the security solution to get information is through inspecting the low-level data sources it controls. The low-level data need to be reconstructed to higher-level semantic knowledge about the guest operating system. Even though the hypervisor sees everything, it lacks understanding of the guest operating system structures. In literature, this problem is known as the *semantic gap*. There are different approaches to bridging the semantic gap. [3]

The semantic gap problem is highly similar to the problems in forensic memory analysis. In forensic memory analysis, the goal is to extract relevant forensic information from memory dumps. Memory dumps are essentially snapshots of the running virtual machine state. They contain exactly the same bits that were present in the memory at the moment the memory dump was taken. The biggest difference between the two is that forensic memory analysis operates on memory dumps that are static and do not change. VMI on the other hand works with live machines, which means that the contents of the memory are constantly changing. Still, some of the work that has been put into analysing memory dumps can be used to help solve the semantic gap. [14]

The semantic information that the guest OS maintains includes, but is not limited to, processes which are currently active (process state is not terminated), the process or processes which are currently running in the CPU(s), modules and DLLs that are loaded into a process and open network connections [15]. The data structures that are defined by the operating system represent these higher-level entities. The hypervisor is not aware of the OS specific information such as the kernel data structures and symbols. Obtaining the OS-level semantic information is crucial for all security analysis as it gives context to all bits and bytes that are seen in the memory. The trustworthiness of the security solutions depends on the correctness of the OS-level semantic information, which is also the reason why bridging the semantic gap is such an important task.

4.2 Bridging the semantic gap

The state of the virtual machine consists of all CPU registers, volatile memory, stable storage et cetera. The hypervisor has no semantic knowledge of this state. The same memory sections or CPU registers can be used for entirely different purposes on different operating systems. The hypervisor does not know wherein the kernel address space critical data structures are located and what is their structure. Applying this semantic knowledge about the monitored system is called bridging the semantic gap. The representation of the virtual machine after the semantic knowledge has been added is called *view of the virtual machine*. [16]

There are four different approaches to bridging the semantic gap – in-VM, out-of-VM delivered, out-of-VM derived and hybrid [3]. VMI techniques fall in to one of these categories based on where from and how the semantic knowledge is received. The out-of-VM techniques are considered to be the true VMI techniques as they do not require any in-VM agents. The semantic gap is bridged entirely outside the monitored virtual machine. [3]

4.2.1 In-VM

In-VM technique is a technique where the semantic knowledge is taken from within the monitored host. The monitored virtual machine contains an agent that communicates with the hypervisor and sends it high-level information about ongoing events state. As the agent runs inside the monitored VM, it is aware of items such as processes and files. [3]

In-VM techniques lack the desirable traits of out-of-VM techniques – evasion resistance, full visibility and tamper resistance – but they avoid the semantic gap issue altogether. The main design challenge becomes ensuring secure coordination and communication between the in-VM agent and the hypervisor. The in-VM agent is exposed to attacks as it resides inside the monitored host. The agent's visibility is reduced as it must rely on APIs provided by the operating system. The in-VM techniques are not in the scope of this thesis. [3]

4.2.2 Out-of-VM delivered

Out-of-VM techniques are VM techniques that solve the semantic gap at the hypervisor level. They are considered to be the true VMI techniques. Out-of-VM techniques have all the benefits of VMI. They are resistant to tampering and have full visibility over the monitored host. The two main out-of-methods to bridge the semantic gap are either to deliver (out-of-VM delivered) the OS dependant semantic knowledge to the security solution or deriving from sources such as hardware architecture. No in-VM agent is used to get the semantic knowledge. [3]

Out-of-VM delivered techniques are the techniques that rely on delivered semantic knowledge about the monitored host. Knowledge about the operating system data structures are delivered to the hypervisor and that delivered knowledge is used as templates to interpret the observed bits and bytes. [3]

Out-of-VM delivered techniques cover early and passive VMI techniques. Information about the guest OS internals (data structures, kernel variables and so on) and their locations and definitions are explicitly told to the security solution. In practise, the OS reliant semantic knowledge is either included in the VMI system, extracted from the monitored OS's source code or obtained through kernel symbols [3]. These solutions are highly reliant on the monitored guest OS. Every operating system has slightly different internals and data structures (as can be seen from Chapter 2, where I briefly looked into kernel data structures that are related to processes and threads). There are even differences between operating system versions. Knowledge of the guest OS internal structure needs to be brought to the VMI security solution for every different OS and OS version that is being monitored. Solutions that rely on memory forensics tool *Volatility* to bridge the semantic gap fall into this category.

Livewire which is the first VMI solution ever suggested (Garfinkel and Rosenblum in 2003), falls into this category. It was designed to detect intrusions in Linux guests that are running on top of VMware workstation. The tool gets the semantic knowledge about Linux data structures by using an adapted version of a Linux kernel dump analysis tool *crash*. Livewire compares the information obtained from the modified version of crash to information obtained from native tools (such as *ps* or *netstat* within the guest). Any inconsistencies between the two outputs would be flagged for closer inspection. [3]

Out-of-VM delivered techniques have some additional drawbacks. The techniques assume that the guest OS data structures do not change at run-time. Normally this is the case, but some kernel rootkits are capable of doing Direct Kernel Object Manipulation (DKOM) and Direct Kernel Structures Manipulation (DKSM). DKSM attacks modify the syntax of the kernel data structure definitions to make the semantic knowledge that is delivered to the security solution obsolete. [15][3]

DKOM attacks directly modifies values (most often pointers) in kernel data structures to hide the malicious activity and mislead the security tools. For example, in Windows environments, the malware can modify the linked process list (each EPROCESS structure has a link to next and previous process) so that the malicious process is detached from the list but remains in the scheduler. The process continues to be running but will not be discovered by tools that work by walking the process list. There are tools that can detect such hidden data structures by scanning through the kernel's memory space (for example, Volatility's *psscan* plugin [7]). [15]

4.2.3 Out-of-VM derived

Out-of-VM derived techniques are techniques that utilize knowledge about the underlying hardware architecture to derive semantic knowledge about the monitored host [3].

Modern hardware components provide modern operating systems functionalities that help them do operating system duties such as memory management. The out-of-VM derived techniques inspect and interpret hardware states in order to get the view of the monitored monitored guest. By relying on the hardware architecture, the techniques are not vulnerable to DKOM or DKSM as the guest operating system needs comply with the virtual hardware. Certain hardware registers specify the location of other crucial data structures. All programs running on the hardware need to comply with the rules set by the hardware. They cannot, for example, change the meaning of the Interrupt Descriptor Table Register (IDTR) to something else. [3][2]

The amount of information that can be derived solely based on the hardware architecture is limited. This information can be vastly increased by combining the derived method with some delivered knowledge about the monitored guest OS. The knowledge that is delivered to the VMI system needs to be *rooted in hardware* so that we still maintain robustness against circumvention. The hardware component the delivered information rooted in is called the *anchor*. [16]

The information is said to be hardware rooted, if there is a memory reference chain starting from a virtual hardware specification to a critical OS data structure. An attacker cannot modify hardware rooted data structures unnoticed nor can he perform his actions at a lower level in order to hide his activities as the view of the virtual machine is built up from the lowest possible level [16]. [3][2]

For example, the system call table can be rooted in hardware in the following way. A x86 architecture processor has a special register IDTR (Interrupt Descriptor Table Register), which contains the base address of the Interrupt Descriptor Table. The Interrupt Descriptor Table is then used to find the System Call Dispatcher and then the system call table, which contains all system call handlers. This combines derived component (the hardware register) with delivered semantic knowledge about the guest OS (the System Call Table). [16]

The fact that out-of-VM derived techniques utilize hardware functionality makes those techniques OS-agnostic. Any operating system that supports the underlaying hardware can be run on the same VMI system (requiring only minor configuration changes).

Hardware anchors

Pfoh et al. [16] identified hardware features from the x86 architecture that can be used as hardware anchors to derive semantic knowledge about the monitored virtual machine.

The CR3 register holds the address of the top-level page directory for the process that is currently in execution. Since each process has their own virtual address space and thus also unique virtual-to-physical page mapping, the top-level page directory is unique for each process. Because the top-level page directory is unique for all processes, its physical address is also unique, which allows process enumerating based on the contents of the CR3 register. The CR3 register would act as process identifier. The system can be set to trap on writes to the CR3 register, which in turn causes the system to trap on context switches (process in execution changes), which in turn leads to the hypervisor being aware which process (which unique CR3) is currently in execution. [16]

Many rootkits place interrupt hooks to the compromised system. Such hooks are generally placed inside the system call mechanism, but they could be placed to catch any interruption within the system. Malware can hook the system call mechanism by modifying the interrupt descriptor table (IDT) to point out to a malicious copy of the system call table, modifying a single system call handler, modifying the IDTR (which contains the size and location of the IDT) to point out to a malicious IDT or by modifying the system call dispatcher routines. [16]

Only monitoring the IDTR allows the attacker to modify the system call table or handles undetected. On the other hand, monitoring the IDT and system call tables alone is not enough either as the IDTR could be changed so that the monitored data structures are not used at all. The security solution needs to monitor the entire chain starting from the hardware registers to capture all malicious modification attempts against the interruption mechanism. As not all items in the chain are hardware dependant (system call dispatchers, system call handlers and the layout of the system call table), some delivered semantic knowledge is needed. But because the delivered semantic knowledge is rooted in hardware, the solution is still robust against circumvention. [16]

The Global Descriptor Table Register (GDTR) holds the address and size of the Global Descriptor Table, Similarly, the Local Descriptor Table Register holds the address and size of the Local Descriptor Table. These data structures are used by the segmentation mechanism in the x86 architecture to specify which segments are accessible at different protection levels. However, many modern operating systems do not use the segmentation mechanism to provide protection and isolation and decide to rely on the paging mechanism instead [5]. [16]

The virtual machine must rely on the hypervisor for all input and output as the virtual machine is provided with virtual copies of the actual I/O devices such as the keyboard or the network interface card. As the hypervisor controls the virtual I/O devices, the VMI mechanism can also monitor them effectively. Monitoring I/O from the hypervisor simply means tapping into the I/O feed and interpreting the data. To monitor network traffic per process, one could use the CR3 as process identifier and combine that with the feed from NIC. The process and network information would be rooted in hardware and thus would also be robust against circumvention. [16]

4.3 Trapping system calls

This section will cover techniques for trapping system calls to the hypervisor.

The Intel's interruption mechanism has 32 slots for system interrupts in the Interrupt Descriptor Table (offsets 0-31) and 224 slots for user interrupts (offsets 32-255). The system interrupts contain interrupts such as Page Faults and General Protection Faults. Any system interrupt can be trapped and the execution can be transferred to the hypervisor. Intel's virtualization extensions do not natively support trapping of user defined interrupts (interrupt code between 32 and 255), but AMD's extension does. To trap user defined interrupts such as system calls on Intel processors, extra steps need to be taken. The state of the virtual machine (virtual hardware) is modified so that the user defined interrupts cause a system interrupt, which then can be trapped to the hypervisor. [17][16]

To make user defined interrupts trap on Intel devices, one can virtualize the IDT and manipulate the IDTR register (and prevent all further modification to it). The IDTR contains the location and the size of the IDT. Each entry in the IDT is 8 bytes long and the 32 first offsets cover the system interrupts. By setting the size of the IDT in IDTR to be 255, all user interrupts cause a general protection fault, which can then be trapped. The next step is to differentiate between naturally occurred protection faults and those that were created by the modification, which can be done by checking the current instruction and whether the interrupt number is above 31. The instruction will be *int n*, where n is the interrupt number if the protection fault was caused by the modification. If the protection fault occurred naturally, the guest is allowed to continue. If the protection fault was caused by a user interrupt, the hypervisor needs to emulate it. [2][16]

System calls may also be implemented using SYSCALL and SYSRET instructions. The instructions rely on Machine Specific Registers (MSR). This system call mechanism can be disabled by unsetting the SCE flag in the Extended Feature Enable Register. If the SYSCALL/SYSRET mechanism is disabled (flag is not set) and the CPU encounters a SYSCALL or SYSRET instruction, it will raise an invalid opcode exception, which can be trapped to the hypervisor. Differentiating between legitimate opcode exceptions and those caused by the modification is simple as if the instruction that caused the exception is not SYSCALL or SYSRET, the exception occurred naturally. If it is indeed SYSCALL or SYSRET, the hypervisor must emulate the instruction. [2][17]

SYSTENTER and SYSEXIT instructions are the third way system calls could be implemented. SYSENTER and SYSEXIT use SYSTENTER_CS, SYSENTER_ESP and SYSENTER_EIP MSRs. The values of the registers are copied into system registers. The value of SYSENTER_ESP is loaded into RSP and it contains the stack pointer for the privilege level 0 stack. The value of SYSENTER_EIP is loaded into RIP and it references the first instruction in the selected procedure or routine. The value of SYSENTER_CS is loaded into the CS register and contains the segment selector for the level 0 code segment. Trying to load the CS register with a null selector causes a general protection fault, which can be exploited to make system calls implemented with SYSENTER/SYSRET to trap. [17]

To make SYSENTER/SYSRET based system calls to trap to the hypervisor, the value of the SYSENTER_CS MSR is saved in the hypervisor and loaded with null. Whenever a SYSENTER executes, it loads a null value to CS and causes a protection fault, which can be trapped. Differentiating such protection faults from legitimate protection faults is a matter of checking if the current instruction is SYSENTER. If the instruction is SYSEN-

TER (or SYSRETURN) the hypervisor needs to emulate the instruction. [2]

Changing the state of the virtual machine to trap events that would not normally cause a VM exit can potentially expose the presence of VMI to a malicious entity inside the VM. The malicious entity can search the system for anything unusual (such as the unexpectedly small size of IDTR). However, the x86 virtualization extensions allow to trap access to most system registers, which allows the hypervisor to present a false value and keep the changes hidden from the guest. [16]

Often it is advantageous to cause the VM to trap to the hypervisor even though there is no native way to do it. One such example is causing the system calls to trap to the hypervisor on Intel machines. When doing so one needs to consider the false positives (traps to the hypervisor that occur because of the modification but are not of interest from the point of view of the VMI). A VM exit is an expensive operation that will cause a significant performance overhead if false positives occur frequently. [16]

4.4 Paging system

Paging system can also be leveraged for VMI purposes. There are three different Page Faults that can be leveraged: Page Not Present, Page Illegal Write and Page Illegal Execute exceptions. [16]

Page Not Present exception occurs when a page is requested for which the page-present flag in the page table is not set indicating that the requested page is not currently present in memory. The exception can be used to indicate any type of access to a particular page. The hypervisor will unset the page-present flag for the monitored page. Any access to that page will cause a page fault, which the hypervisor can then trap. Ether uses this approach to monitor system calls as they set the MSR which contains the offset of the system call dispatcher to point out to a page which is set be not present. All system calls that use the system call mechanism can then be trapped with the resulting Page Not

Present exception. [16]

Page Illegal Write exceptions can be used to monitor write access on specified pages. By revoking write permissions on the pages (unsetting write flags) any write attempt on the page will cause a Page Illegal Write exception which the hypervisor will then trap. This feature can be particularly useful when protecting static data structures or code segments. By using Page Illegal Write exceptions to track write access on protected areas of memory, we are using the page system's protection mechanism as they are intended to be used (even though they are used from outside the VM). If the system uses shadow page tables, the changes are not visible from within the VM as shadow page tables are a structure the hypervisor maintains and the guests do not have access to it. [16]

Paging mechanism could also be used for code execution trapping with the Page Illegal Execute exception. Similarly to other the previous cases the trap is achieved by unsetting the flag stating execute permission for the monitored page. Any execution attempt would then cause Page Illegal Execute exception which the hypervisor then traps. However, there are few major issues that lead to making this approach non-suitable for many applications. [16]

Assuming we are trapping execution of a page using the Page Illegal Execute exception and the page contains a function. When the function is called, the Page Illegal Execute exception is raised at the first instruction and a trap to the hypervisor occurs. The hypervisor then emulates the instruction and the execution moves on to the next instruction, which again causes PIE exception and execution is transferred to the hypervisor. The execution is transferred to the hypervisor on each instruction and we end up single-stepping over the page, which introduces considerable performance overhead.

To avoid single-stepping we would need to reset the execution flag once the first instruction gets trapped. This introduces additional issues. Now the hypervisor needs to know when it can unset the execution flag again to trap any further access to the monitored page. To do this the hypervisor would need to constantly poll the VM state or unset the flag after some time has passed. Also, once the flag has been set to allow execution, any other process could start executing the protected function without the hypervisor becoming aware.

5 Implementations of VMI

5.1 Livewire

5.1.1 VMI-based Intrusion Detection System

Livewire is the first VMI-based security solution ever proposed. It was designed by Garfinkel and Roseblum and introduced in their paper *A Virtual Machine Introspection Based Architecture for Intrusion Detection* [18]. The purpose of the design was to allow the IDS to have the benefits of both network and host-based systems without gaining the weaknesses of either type. By placing their IDS on the hypervisor, Garfinkel and Rosenblum managed to have the visibility of a host-based IDS and the resilience of a network-based IDS. [18]

By placing the IDS on the hypervisor, the solution must solve the semantic gap issue as it now lacks higher level knowledge about the monitored host. Livewire solves the semantic gap issue by using knowledge of the operating system structures inside the virtual machine to interpret the low-level hardware events [18] (out-of-VM delivered strategy). When the semantic gap issue has been solved, the IDS policies can be written as high-level statements about entities inside the monitored host.

In addition to having the benefits of both IDS types, a VMI-based is much safer in case of system failure or crash. When a network-based IDS fails, all network traffic passes through without the IDS becoming aware. Stopping the network traffic whenever the IDS is down is not an option as network mediums are often shared among multiple hosts and suspending the network traffic would lead to a significant denial-of-service risk. Host based IDSs do not fail much better than network-based IDSs. If the IDS is located at the user-level, suspending system activity while it reboots cannot be done as the IDS itself relies on the OS to restart it (unless the IDS only monitors a single application). When a kernel-based IDS crashes, it often causes the entire system to crash or allows the attacker to compromise the OS kernel. When a VMI-based IDS crashes, the virtual machine it monitors can simply be paused while waiting for the IDS to reboot. [18]

As Livewire relies on knowledge about the monitored OS architecture, it is vulnerable against attacks that modify how OS data-structures are being used. If the malware manages to successfully monitor how OS data-structures are used, the view of the monitored system for the IDS solution becomes false and the malware can evade detection. Rooting information in hardware as discussed in Section 4.2.3 is a way to mitigate this issue. [18]

5.1.2 Design of Livewire

When designing Livewire, Garfinkel and Rosenblum wanted to minimize changes to the hypervisor in order to avoid creating additional bugs, which could then potentially be exploited by attackers. To provide additional functionality for their VMI-based IDS they leveraged existing functionality in the hypervisor. Another important design goal in their project was to minimize performance overhead caused by inspecting the state of the virtual machine. To minimize performance overhead, they avoided trapping events and interrupts that would occur frequently and instead it focused on monitoring events that would imply misuse (monitoring memory segments that should not change at runtime). [18]

They also pointed out that exposing the internal state of the monitored VM potentially also exposes the VMI-based IDS, and thus also the hypervisor for attacks. They reduced the risk of an IDS compromise leading to the hypervisor becoming compromised by separating the IDS from the hypervisor. [18]

Communication between the hypervisor and the VMI IDS is achieved by sending

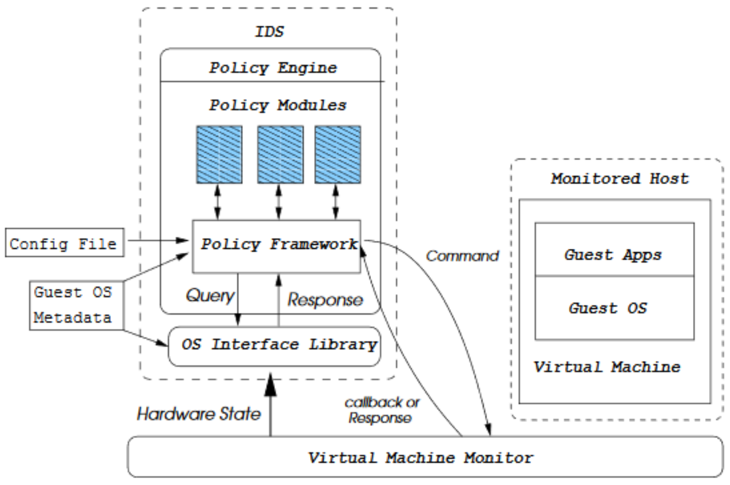


Figure 5.1: Image illustrating the architectural structure of Livewire. The IDS sends commands to the hypervisor, which replies with information about the internal state of the monitored host. The OS Interface library is used to give meaning to the low-level state information provided by the hypervisor

commands over a designated communication interface. Livewire has three types of commands: inspection, monitoring and administrative. Inspection commands are used to examine the state of the VM. State information that could be examined consists of memory and register contents and I/O devices' flags. When the hypervisor receives these commands, it replies with the value for the requested query. [18]

Monitor commands are used to find out when monitored machine events occur and receive a notification for it. Monitored events can for example be changes in privileged registers or changes in certain memory range. The hypervisor sends a reply to the VMI IDS whenever an event specified by the monitoring command occurs. [18]

Finally, administrative commands are used to control the execution of the monitored VM (resume, pause, checkpoint and reboot). [18]

The VMI IDS consists of an OS Interface Library and a policy engine. The OS Interface Library uses knowledge of the monitored guest OS to interpret the virtual machines low-level state information. The library contains the delivered OS specific semantic knowledge required to interpret low-level hardware state. In Livewire's case the library contained semantic knowledge about Linux guests. Livewire's OS Interface Library is built by modifying a Linux crash dump analysis tool *crash*. [18]

The policy engine is the element which is responsible for determining whether the monitored guest VM has been compromised. It consists of a policy framework and policy modules. The policy framework allows the user to write high level policy modules to implement the security policy. Livewire included six different sample policy modules. Four modules are polling modules that are run periodically and two are triggered by specific events. There is a major weakness with polling modules as malware could potentially do what it intended to do and hide during polling cycles leaving it undetected. [18]

One example of a sample policy module in Livewire is a module that detects hidden elements inside the guest. It polls the in-VM view of the guest via remote shell and commands such as ps, ifconfig and netstat. Simultaneously it inspects the VM state from the hypervisor and reports back any inconsistencies between the two views. A potential issue for such a module is that the views might not be constructed exactly from the same point in time. [18]

5.2 Ether

5.2.1 A Transparent External Malware analyzer

The motivation for Ether stems from the observation that understanding what occurred in a compromised asset is just as valuable as knowing the asset was compromised. To understand the true behavior of modern malware, the analyst needs to get an unobstructed view of the malware. Malware authors do, however, attempt to complicate malware analysis by including anti-debugging, anti-instrumentation and anti-VM techniques to hinder runtime observation of malware. [4]

Ether attempts to be a transparent external malware analyzer, which means that it cannot induce any side-effects that are detectable by the malware it is studying. When the malware analyzer is transparent, it can obtain an execution trace that is identical to the same program being run without the analyzer being present. If the transparency requirement is not met, the malware can learn the presence of a malware analyzer and modify its behavior. To meet this requirement, Ether resides entirely outside the monitored guest. [4]

Ether's approach diverges from other malware analyzers as it does not contain any in-guest software components, nor does it rely on API virtualization or emulation. Instead, it relies on hardware virtualization extensions such as Intel VT. Virtualization extensions were chosen as the analysis mechanism because they do not interfere with the original instruction stream, CPU stream or memory state and thus meet the transparency requirement. By relying on hardware virtualization extensions, Ether is available to remain entirely outside the monitored virtual machine. [4]

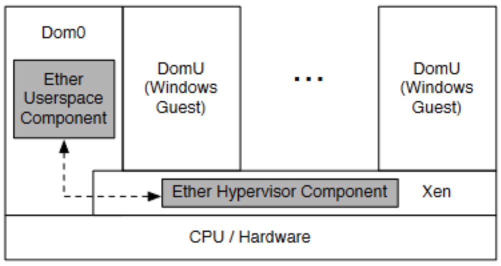


Figure 5.2: Architectural overview of Ether. Ether contains an userspace component in dom0 and a hypervisor component. The hypervisor component is used to detect events from the monitored guest and the userspace component is used to determine which processes and events should be monitored. [4]

5.2.2 Implementation of Ether

Ether is based on the Xen hypervisor version 3.1.0. Xen was selected as the hypervisor because it is a mature open source project, and it has existing communication mechanisms that could be leveraged for Ether. Ether relies on Intel's VT extensions and supports Windows XP SP 2 guest operating systems. The researchers chose Windows XP as at the time of implementing Ether, Windows XP was the most common PC operating system. [4]

Xen hypervisor is a type 1 hypervisor and it runs at the most privileged layer. It has two different domains – Dom0 and DomU. Dom0 is the priviledged domain that gets started first. It is used to manage the unprivileged guest VMs in domU and perform administrative tasks on the Xen hypervisor. It is the only domain that can directly access hardware. The unprivileged domain, domU, contains the guest VMs that are being analyzed. In Ether's case the guest operating systems are Windows XPs with Physical Address Extension and large memory pages disabled. They modifications to the default XP installation are made to make memory write detection easier. [4]

Ether has a hypervisor component and an userspace component which is located in the administrative VM (dom0). The hypervisor component detects events such as system call execution and memory writes in the analyzed VM. The userspace component is responsible of determining which processes and events in the guest should be monitored. The userspace component is also responsible for bridging the semantic gap and interpreting the low-level information. It can for example convert the system call number into a system call name and display the system call arguments. [4]

Ether is capable of monitoring instruction and system call execution in the guest. It can also detect any memory write events made by a guest process. [4]

Instruction execution monitoring is achieved by making sure each instruction causes a debug exception, which is then handled by the hypervisor. The debug exception is caused on every instruction by setting a trap flag in the monitored VM. Once it receives the debug exception and handles it, it sets the trap flag for the next instruction. By doing so it manages to trap to the hypervisor on every instruction and allows Ether to execute the target process one step at a time. [4]

Memory writes are monitored by using shadow page tables and handling guest's page faults. Ether causes a page fault to occur on all memory writes by removing the write permission for shadow table entries. Once Ether receives a page table fault, which is caused by the modification, the userspace component is notified, the write permission is reset and the instruction which caused the fault is re-executed. When the instruction has finished executing, write permission is again removed from shadow page table entries. Ether ensures that the page table faults caused by the modification do not reach the guest to ensure the guest does not gain any indication of its presence. All page faults that occur naturally are forwarded to the guest. [4]

Ether traps system call execution through SYSENTER/SYSEXIT instructions. It forces all system calls invoked using SYSENTER mechanism to cause a page fault by setting the value of SYSENTER_EIP_MSR to point out to a page that is guaranteed not to be present in memory. Normally the register would point out to a pre-defined address in kernel. When the SYSTENTER instruction is run, the value of the instruction pointer is changed to match the value of the MSR. The original value of the MSR is saved in Ether's memory. Now whenever a SYSENTER instruction is ran, it will cause a page fault at the chosen address and Ether knows the guest attempted to run a system call. The value of instruction pointer is then changed to match the expected value and the execution of SYSENTER can resume. The arguments, return address and return values are gathered by performing memory reads on the guest. Ether uses knowledge of the internals of Windows XP SP2 to know where to look for these entities. [4]

Ether allows to limit the analysis scope per process. This is done by monitoring for context switches. Whenever the process currently in execution is switched, Ether checks whether the upcoming process is the monitored process and the either enables or disables analysis. Processes can be specified either by using the process name or their top-level page directory address (the value that gets set into CR3 when the process enters execution). [4]

5.3 Nitro

5.3.1 Hardware-based system-call tracking mechanism

Nitro is a hardware-based system-call tracking and monitoring platform. It consists of an userland component – called Nitro – and a set of kernel modules that allow system call trapping on KVM guests. As the monitoring solution is outside the monitored VM, it is also isolated from malicious activity occurring inside the VM and thus remains hidden from the guest OS. It is the first VMI-system that supports capturing all three system call mechanisms provided by the Intel x86 architecture and it has been tested on Windows and Linux (32 and 64 bit) guests. Data is captured in real-time without diminishing the usability of the guest. Nitro uses hardware anchors to prevent the malware from evading monitoring. [2]

The key properties of Nitro are guest OS portability and evasion resistance. Guest OS portability refers to the property which allows Nitro to be used on different guest Oss without requiring major changes to the system. To achieve OS portability, the VMI solution cannot rely on knowledge of the guest operating system, but on knowledge of the underlaying hardware architecture. [2]

A mechanism is said to be evasion-resistant if it is impossible for an attacker to circumvent the monitoring mechanism when it is correctly implemented and deployed in an ideal system. The term ideal system implies that there are no flaws or errors in the design, which in practise is not often the case. However, the mechanism can only be circumvented if and only if such a flaw is successfully discovered and exploited by the malicious entity. The fact that Nitro is a hardware-based monitoring solution is crucial for achieving this property. [2]

When the VMI monitoring solution bridges the semantic gap in order to understand the low-level activity of the guest, it requires either knowledge on the underlying hardware or knowledge on the monitored guest OS. When the VMI solution chooses to rely on knowledge of the guest OS, it is vulnerable against attacks that that change how the operating system uses particular data structures, because knowledge of the guest OS is not bound to the running OS kernel. As discussed in earlier chapters (see Chapter 4), such attacks are not applicable when knowledge is rooted in hardware (out-of-vm derived). A VMI-mechanism is then evasion resistant if two requirements are met: The monitored portions of the VM's state must be rooted in hardware and each piece along the chain (chain from the hardware anchor to the monitored portion of the VM's state) also needs to be monitored so that all modification attempts are discovered. [2]

5.3.2 Implementation of Nitro

Nitro is based on KVM, which is split into two portions - The user application, which is built on top of QEMU and a set of Linux kernel modules. The user application portion has the QEMU monitor, which is used to control the monitored VM (pause and resume VM, read VCPU register values and so on). The Nitro modified the KVM to include new commands that are used to control Nitro's features. The new commands are all input through the QEMU monitor. [2]

The commands that are given to the QEMU monitor are sent to the KVM's kernel module portion through an I/O interface. The majority of the initial work surrounding Nitro was around the kernel module's which introduced VMI capabilities to the KVM hypervisor. The userland component is used to communicate the low-level events to the user and then interpret them. [2]

Nitro is capable of trapping all three methods of invoking system-calls. The three system call mechanisms provided by the Intel x86 architecture are Interrupt-based, SYSCALL-based and SYSENTER-based. The implementation of the trapping mechanisms is done as described in section 4.3. [2]

Nitro uses the value of the CR3 register as process identifier. When running Nitro without a backed, the CR3 register will be the only indicator of which process caused the event. The value of CR3 register is enough to uniquely identify the process which caused the system call. To get the process ID and process name, Nitro needs to be run with a backend that provides higher level information. [2][19]

Nitro can be used in conjunction with different operating system specific back-ends that transform the low-level events into higher-level ones by analysing the OS memory. For example, when a system call is captured, it transfers the collected CR3 register into PID and process name and displays the name of the system call that has occurred. To achieve this task Nitro uses libVMI library. To run Nitro with a backend, the user needs to install libVMI in addition to Nitro. Because it uses libVMI, will need the configuration files in described in Section 6.2.1. [19]

The default output in a Nitro event consists of the value of CR3 register, value of rax register (which contains the system call number), system call type (syscall or sysenter), the vcpu number, and the direction of the system call (enter or exit) and it is obtained via the *as_dict* function, which outputs the Nitro event as a dictionary. The user can display additional information, such as the value of other machine registers, for each event. The default output of Nitro is stdout, but an output file can be specified. [19]

In Nitro, the user space listener is notified before the system call has been executed, which allows to create handlers that step over and run its code possibly modifying the return values and the values of machine registers. The user can set a system call filter so that only a subset of all system calls get trapped or they can trap every system calls. [19]

5.3.3 Future direction

Nitro is no longer being maintained but the development effort of giving VMI capabilities for the KVM hypervisor continues on KVM-VMI project, which evolved from Nitro. The KVM-VMI project contains a new set of KVM patches developed by BitDefender, which provide the KVM hypervisor with introspection capabilities and a new VMI API called KVMi. KVM-VMI is the project that is set up when using LibVMI with the KVM hypervisor. [19]

5.4 DARKVUF

DRAKVUF is a modern dynamic malware analysis system which utilizes virtual machine introspection to remain isolated and hidden from the monitored malware. The goal of DRAKVUF to have a scalable, automated platform that is capable of efficiently analysing the runtime behaviour of the monitored virtual machine. [20]

5.4.1 Implementation

DRAKVUF is built on top of the Xen hypervisor. It runs in the control domain (dom0) of the hypervisor and does not need any in-VM components that could expose its presence to the analysis target. To get virtual machine introspection related functionality, such as direct memory access, DRAKVUF relies on LibVMI library. [20]

One of the design goals of DRAKVUF is to have a scalable malware analysis platform and maximise the throughput of malware samples. DRAKVUF's approach to solve this revolves around running concurrent analysis environments simultaneously while minimizing the hardware requirements. DRAKVUF can be run in a mode where it creates clones of the analysis VMs and runs different malware samples on those clones. The mode leverages Xen's copy-on-write memory interface and Linux Logical Volume Manager's (LVM) copy-on-write disk functionality. When this functionality is used, concurrent analysis instances can share large portions of memory. According to the tests performed by a DRAKVUF's authors the copy-on-write saves an average of 62.5% of memory. [20]

In order to transfer control to the hypervisor during interesting events, DRAKVUF relies on breakpoint injection. A breakpoint instruction (INT3) is automatically injected into code sections inside the VM's memory that are thought to be interesting. The CPU is configured to cause a VMEXIT whenever the breakpoint instruction is executed, and the hypervisor is instructed to forward such events to DRAKVUF. By relying on the breakpoint instruction, DRAKVUF can trap the execution of both kernel and userland code. DRAKVUF is the first to apply breakpoint injection for execution tracking of the entire operating system. [20]

A common approach for execution tracing of a process is to monitor the system calls the malware issues. By monitoring only system calls, the execution trace is limited to showing the interaction between the OS kernel and the user-space program. It does not include kernel-mode rootkits. DRAKVUF directly traps the internal kernel functions via breakpoint injection, which allows it to also monitor malicious drivers and rootkits. The location of the kernel functions is determined by extracting information from the debug data given by the kernel using a forensic tool called Volatility3. Volatility3 is used to create the json profile of the OS kernel. [20]

DRAKVUF automatically locates the OS kernel at runtime by making use of the discovery that Windows 7 uses FS and GS registers to store a kernel virtual address which points out to _KPCR structure (Kernel Processor Control Region). The _KPCR structure is always loaded into a fixed relative virtual address within the kernel. That relative virtual address is identified by the kernel symbol KiInitialPCR. The base of the OS kernel is then found by subtracting the value of KiInitialPCR found by Volatility from the address in the vCPU register. Once the kernel base address is found, DRAKVUF can use breakpoint injection to trap all kernel functions. [20]

DARKFVUF addresses a previously ignored issue of starting malware samples. Previously a malware sample needed to be manually started or started by an in-guest agent or script. Manually starting malware samples hinders scalability and using in-VM guests (or autorun scripts) can potentially expose the presence of the monitoring environment. Instead, DRAKVUF hijacks (the hijacking method is explained later) an arbitrary process to start the execution of the malware sample. [20]

The process that DRAKVUF can hijack needs to have a windows kernel library kernel32.dll loaded. DRAKVUF hijacks a process by first trapping all context switches (write operations to CR3 register). When DRAKVUF captures such an event, it inspects which libraries are loaded by the new process. The inspection is done by walking the list of loaded modules within the process. If the process has kernell32.dll loaded, execution will be switched into single-step mode until the process starts executing user level code. [20]

The hijack mechanism starts at the first user-level instruction executed by the process. The mechanism locates CreateProcessA routine from the kernel32.dll's export table. It then modifies the stack to contain the parameters required to call the function. The original instruction pointer is pushed on the stack as the return address. A breakpoint instruction is also injected into the return address to indicate when the routine has finished. The address of the CreateProcessA is placed on the instruction pointer and the execution is resumed. [20]

Once the function has finished running, the PID and handle information of the newly created process is extracted by examining the RAX register (register used to store integer or pointer return values [21]). Then the vCPU values and stack are restored to match the state before the hijack occurred and the hijacked process continues executing normally. [20]

Malware often uses external resources and input to function properly. One example of such a case is a malware attempting to connect to a remote CC server. Failing to connect to the server can be seen by the malware as an indication that it is being monitored. However, legislation poses some limitations on running malicious code. Infected host cannot be allowed to infect other systems if the administrator is aware of the malware infection. Thus allowing full network access to the analysis target is not often possible. [22]

DRAKVUF can monitor Windows heap allocations to mitigate DKOM attacks that unlink structures from linked lists. The traditional forensics approach for discovering the

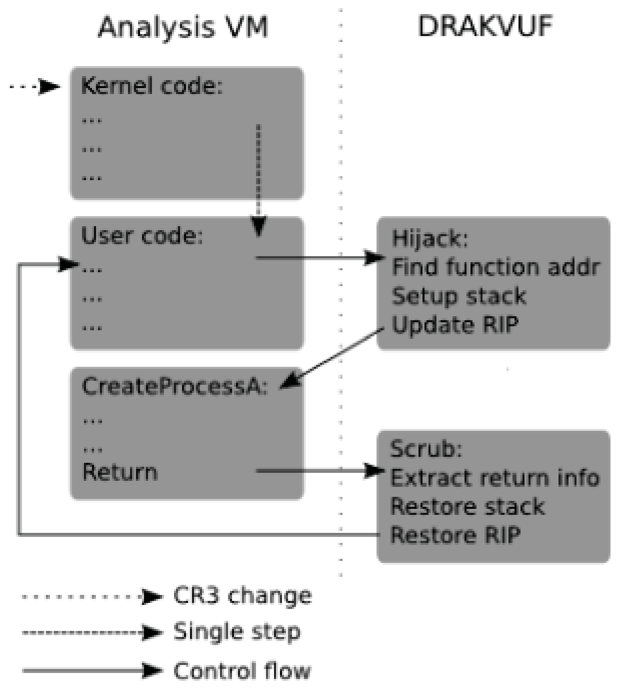


Figure 5.3: Image illustrating the execution flow of DRAKWUF's process hijacking. The hijacked process is used to launch the malware sample. After the new process has been started the state of the hijacked process is restored. [20]

hidden structures is to scan the physical memory for _POOL_HEADERS which are attached to objects in the Windows kernel heap. The header contains a four-character tag that describes the type of the structure. The traditional approach to discovering unlinked structures in the kernel focuses on scanning for such structures. The anti-forensics tricks used by malware to mitigate pool tag scanning is to try to overwrite the headers or to change the size of the structure so that it gets placed into the *big page pool* where structures do not contain such headers. DRAKVUF tackles this issue by injecting breakpoint instructions to Windows kernel functions responsible for heap allocations. This allows DRAKVUF to be aware of the location of all structures within the kernel heap. [20]

DRAKVUF uses the heap allocation mechanism to track filesystem access. Whenever a file is accessed, a _FILE_OBJECT is allocated within the kernel heap. The structure is then used to determine the name, path and access permissions of the file. [20]

DRAKVUF has the capability to carve deleted files from memory whenever one is removed. This functionality is desirable as many malware droppers create multiple temporary files that are used during the infection process. However, malware droppers tend to clean up after themselves and remove all files they create. DRAKVUF carves the removed files directly from memory using Volatility. [20]

The process of carving deleted files starts by first intercepting system calls that are used for file deletion. Once such call has been intercepted, DRAKVUF looks at the arguments passed to the function which contain file handle information. The file handle information contains a reference number to an entry in the owning processes handle table. That number is used to find the corresponding entry in the handle table, which is used to find the correct _FILE_OBJECT structure. Once the structure has been found, Volatility can be used to extract the file. [20]

5.4.2 Usage

DRAKVUF is used via the command line interface by launching the compiled C++ file located in DRAKVUF's src folder. DRAKVUF has two mandatory switches, -r and -d, that need to be provided when it is launched. The -r switch is used to tell the path to the OS kernel's json profile. The -d is used to indicate the domain id of the domain being analysed. The domain id is found by running Xen's *xl list* command.

The OS kernel json profile is created using Volatility3's tool *pdbconv*. The value of PDB GUID and the kernel's filename are passed to the tool, which then creates the required json profile pointed out by the output switch. LibVMI's configuration file is then edited so that the configuration entry which belongs to the desired domain points out to the json profile (volatility_ist = path/to/json/profile).

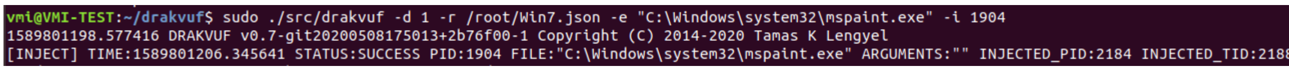
When DRAKVUF is run without any additional switches, it monitors the execution trace of the entire operating system. The execution trace contains information about events such as system calls, heap allocations and file access. The events which are being monitored is determined by activating or deactivating DRAKVUF plugins. Some plugins require additional json profiles to function.

System calls create two entries to the logs. The first entry is logged when a system call is made, and the second entry occurs when returning from system calls. All log rows that are represent system calls or system call returns contain the vCPU number, value of CR3, path to the executable of the process, process, thread and session id, and the system call name and number. Entries that represent system call invocations also contain the number and value of the arguments passed to the system call. Rows that represent system call returns also contain the return value for the system call. The plugin responsible for monitoring system calls is called syscalls. The user can define which system calls should be trapped by passing DRAKVUF a file containing a list of system calls to be trapped.

Poolmon is the plugin used to track kernel heap allocations in Windows. In addition to the normal identifying information such as vCPU number and CR3, the log rows created by Poolmon contain the pool type, size of the allocation and the pool tag.

One example of plugin that requires additional configuration profiles is *socketmon*, which is used to monitor the TCP and UDP sockets for Windows guests. To function properly, the user needs to create a profile for tcpip.sys kernel module, which is commonly located at C:\Windows\System32\drivers\tcpip.sys. The profile is created by first copying the module to the location where the profile is being created (for example to the dom0). Then the Volatilty3 tools pdbguid and pdbconv are used to create the json profile like when creating the profile for the OS kernel. The path to the additional profile is passed to DRAKVUF using switch –json-tcpip or -T.

To inject a process into the virtual machine using DRAKVUF, one needs to first copy the executable to the analysis target. After that, the executable is launched by providing DRAKVUF the path to the executable file using -e and the process ID for the process to be hijacked. An example command is visible in image X.



```
vmi@VMI-TEST:~/drakvuf$ sudo ./src/drakvuf -d 1 -r /root/Win7.json -e "C:\Windows\system32\mspaint.exe" -i 1904
1589801198.577416 DRAKVUF v0.7-git20200508175013+2b76f00-1 Copyright (C) 2014-2020 Tamas K Lengyel
[INJECT] TIME:1589801206.345641 STATUS:SUCCESS PID:1904 FILE:"C:\Windows\system32\mspaint.exe" ARGUMENTS:"" INJECTED_PID:2184 INJECTED_TID:218
```

Figure 5.4: An example of DRAKVUF's output when a new process is injected. In this case process 1904 is hijacked to launch Microsoft Paint. The json profile for the OS kernel is located at /root/Win7.json and the domain ID is 1.

There are several plugins that provide DRAKVUF with additional functionality and monitoring capabilities. Filedelete plugin is used to carve deleted files from memory, exmon to monitor for exceptions in the guest, filetracker to monitor for the use of _FILE_OBJECT structures, memdump for memory dumping, and librarymon to trace library loads. The plugins are not limited to those mentioned here. For full plugin list, see DRAKVUF's github page [1].

[1]https://github.com/tklengyel/drakvuf

6 LibVMI: Virtual Machine Introspection library

6.1 General capabilities

LibVMI is a C library, which allows monitoring low-level details of a running virtual machine by viewing the virtual machine's memory, accessing its virtual CPU registers and trapping on hardware events. LibVMI is an out-of-VM VMI-library, which means that the introspection is done from outside the monitored guest. LibVMI supports Xen and KVM hypervisors and modified QEMU. It is designed to be run on Linux systems (file, Xen or KVM access), but it can also be run on Mac OS X (only file access). [23]

Many memory forensics tools such as Volatility are written in Python instead of C. To take full use of such tools, the LibVMI library has Python wrappers. Having the Python wrappers enables the LibVMI library to be integrated with other memory forensics tools, such as Volatility, that are written in Python.

Each function in the libVMI C-library has a semantically equivalent function in the Python wrapper. In addition, the Python wrapper has an additional function zread, which will read from desired location. The function will never fail. Bytes that cannot be read will be replaced with zeros instead. The function was specifically added to have improved integration with Volatility.

One of the key features of LibVMI from the perspective of memory forensics is that it can be integrated with Volatility, which is a popular tool that has lots of semantic knowledge on modern operating systems such as different Windows distributions and versions. Volatility is normally used to analyse memory dumps, but integrating it with LibVMI enables the possibility of analyzing the memory of a running virtual machine with high semantic knowledge.

To integrate Volatility with LibVMI, one needs to install the Python wrappers for the LibVMI. Inside the package is a folder containing the volatility address space plugin that allows volatility to be ran against the introspected guest. That address space plugin needs to be copied to */volatility/plugins/addrspaces* inside the folder where the code of volatility is located.

Memory forensics is not the only use case of LibVMi. Other use cases include VMI based debugging such as pyvmidbg [24] (a LibVMI-based debugging server implemented in Python) or runtime security solutions such as VMI based firewalls, intrusion detection systems, malware analysis as well as operating system kernel hardening. LibVMI allows single-stepping, which is a powerful tool for debugging and malware analysis as it allows to execute the program one instruction at a time.[23]

Initially LibVMI evolved from XenAccess. The XenAccess project had some capabilities to do VMI on virtual machines running in Xen. It could only support virtual machines with 32-bit operating systems. It also required the developer to manually map guest VM pages, operate on then and then unmap them. XenAccess version 0.5 was the starting point for LibVMI. The project was expanded to support the KVM hypervisor (which requires some additional patches as KVM does not natively have VMI capabilities) and other hypervisors. The LibVMI project also removed the need for the developer to manually map memory pages and introduced read and write operations that automatically walk the page tables of the guest. The project was also expanded to support 64-bit guest operating systems and the Python wrapper was made. [23]

As KVM hypervisor does not natively have VMI capabilities, it needs an additional steps to enable memory access. There are currently two different options to do it. The first option is to install modified versions of QEMU and KVM. The second option is to use KVM VM's GNU Debugger (GDB), which results in a slower connection. The LibVMI library only tequires one memory access technique. The modified QEMU-KVM approach is the primary way of achieving memory access so the LibVMI will first look for it and use it if it is available. If it is not available it will fall back to GDB connection. To gain additional access (such as pausing and resuming the VM), the libvirt library is used. With KVM the choices are to either use a faster connection that is slighlty harder to install or to have a slower connection with easier installation. [23]

Different hypervisors are supported by having a driver specific for that hypervisor. The LibVMI library could be expanded to other hypervisors by writing a new driver for the new hypervisor. When LibVMI is started, it automatically tries to determine the hypervisor that is available and select the correct driver. [23]

Accessing the memory of the guest from outside the guest is a costly operation and error prone if it needs to be done manually. The LibVMI removed the need for the developer to manually manage guest's page tables, but the library still needs to manage them efficiently. To do this, the LibVMI has four different caches three of which handle basic mappings (virtual address to physical address, virtual address to process identifier and kernel symbol to virtual address). The last cache is a page cache. [23]

The page cache handles copying VM memory pages into memory to support the new read and write memory functions. It uses the hash value of the physical page address as page cache entry identifier. The entry contains additional metadata about the memory page and a pointer to the data. It also has a second structure, which slightly resembles a TLB as it keeps recently used hash values on top of the list to make it faster to access them. Whenever the list grows over a predetermined limit, the bottom half of the list is removed. [23]

6.2 LibVMI usage

6.2.1 Initialization

When using the QEMU-KVM combination as the hypervisor, the library will rely on libvirt. Regardless of which access method is used, the VM XML definitions used by libvirt need to be modified. In both cases, the domain field needs to be modified in the same way.

When using GDB connection, the new <qemu:commandline>level is simpler as the only addition is <qemu:arg value='-s'/>, which enables GDB access to the KVM VM. The same result can be accomplished by adding '-s' switch to the VM creation line.

When using KVMi (patched KVM), the XML configuration modifications are longer, which can be seen in figure 5.1. The new <qemu:commandline>level has additional elements, which for example specify the Unix socket path, which is used to access introspected VM. In the C code, the path to the socket is passed to the *init_data* entry *VMI_INIT_DATA_KVMI_SOCKET*.

For each virtual machine that is being introspected, the LibVMI excepts to find an entry in a configuration file. The library expects the configuration file to be found in either *$HOME/etc/libvmi.conf* or */etc/libvmi.conf*. Each entry in the configuration file specifies the operating system of the guest, location of the symbolic information and offsets that are used to access data inside the virtual machine.

Each configuration entry has the the name of the virtual machine (the name that is listed with for example the command virsh list) followed by a set of key-value pairs specific for that VM. There are nine different keys some of which are solely for Windows or Linux based guests. The keys that are required for Linux hosts have linux_ prefixes whereas keys specific for Windows guests have win_ prefixes.

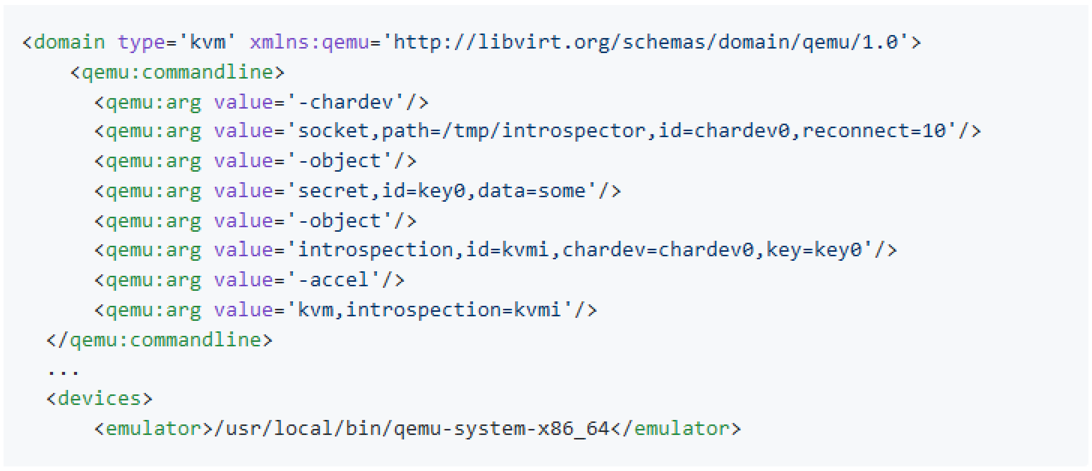
- *ostype* key is used to specify the operating system family. It is required for both operating system types (Linux and Windows).

- *sysmap* key holds the path to the System.map file of the guest being introspected. The System.map file needs to be copied from the VM to the hypervisor. In Linux hosts, the System.map file is generally found under /boot.
- *linux_tasks* tells the offset to task_struct->tasks
- *linux_mm* tells the offset to task_struct->mm
- *linux_pid* tells the offset to task_struct->pid
- *linux_pgd* tells the offset to task_struct->pgd
- *win_tasks* tells the offset to EPROCESS->ActiveProcessLinks
- *win_pdbase* tells the offset to EPROCESS->Pcb->DirectoryTableBase
- *win_pid* tells the offset to EPROCESS->UniqueProcessId

The offsets can be specified either in hex or decimal. Unless told otherwise, the configuration file assumes numbers are decimals, so hexadecimal values should be preceded with ”0x”. LibVMI provides tools that can be used to extract these values from the guests. For Linux guests, the tools need to be copied to the VM and run inside the introspected host.

The configuration information can be passed to the library in three diffent ways. The first and default way of passing configuration data is to use the configuration file /etc/libvmi.config. The configuration information can also be passed in a string (VMI_CONFIG_STRING) or a dictionary (VMI_CONFIG_GHASHTABLE).

LibVMI allows staged initialization with VMI_INIT_PARTIAL flag, which indicates that only a portion of the required initialization data is given. If the flag is set, the LibVMI library will only init enough to view the physical addresses of the introspected host. The intended usage of the partial flag is to first partially initiate the LibVMI instance, run some heuristics against the physical memory of the guest to determine rest of the parameters



```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
    <qemu:commandline>
      <qemu:arg value='-chardev'/>
      <qemu:arg value='socket,path=/tmp/introspector,id=chardev0,reconnect=10'/>
      <qemu:arg value='-object'/>
      <qemu:arg value='secret,id=key0,data=some'/>
      <qemu:arg value='-object'/>
      <qemu:arg value='introspection,id=kvmi,chardev=chardev0,key=key0'/>
      <qemu:arg value='-accel'/>
      <qemu:arg value='kvm,introspection=kvmi'/>
  </qemu:commandline>
  ...
  <devices>
      <emulator>/usr/local/bin/qemu-system-x86_64</emulator>
```

Figure 6.1: Figure showing the libvirt XML configuration changes required to use the KVMi. The xmlns:qemu part is added to the domain tag in both memory access methods. In the qemu:commandline level of the XML the Unix socket path (/tmp/introspector) is specified.

and then call function vmi_init_complete to finish the initialization. After the program has finished runnig vmi_destroy should be called to free up used memory and close any remaining handlers.

6.2.2 LibVMI API overview

Memory reads and writes are handled through various API calls. There are numerous different read and write functions available that read or write different chunks of memory from the provided address which can be physical address, virtual address or a kernel symbol. Other functionality provided by the API are pagetable lookups, address translations, single-stepping, register reads/writes and event listening. The library has declared numerous enums that represent X86 architecture registers that can be interesting from the point of view of VMI.

As LibVMI handles guest page table lookups, the usage of memory read operations is simpler. LibVMI has separate read functions for different chunks of memory (8, 16, 32 or 64 bits) and for different types of memory addresses (virtual or physical). Provided the LibVMI instance is fully configured, it also supports direct kernel symbol reads. The naming of the read functions is consistent as they all have form *vmi_read_<size>_<address type>*.

The exact number and the type of parameters naturally varies between functions. Generally the read functions require the vmi instance and the address that is being read from. Some read functions, such as those that access the virtual memory of a specified process, require additional parameters. For example, the function *vmi_read_64_va* requires parameters vmi_instance, vaddr and pid. The vmi_instance parameter is the initiated vmi instance, vaddr is the virtual address which is being read and pid refers to the process ID of the process whose virtual address is being read. Calling it would read 64 bits from the virtual address pool of the process specified by the process ID starting at the virtual address given as parameter.

In addition to reading a predetermined chunk of bits (8, 16, 32 or 64), LibVMI has read functions for reading an address or a string from the memory. The functions that read an address from the memory behave differently depending on whether the guest OS is a 64-bit or 32-bit system. In 64-bit systems addresses are 8 bytes long so the function reads 8 bytes from the specified address. In 32-bit systems the addresses are only 4-bytes long so the function reads only 4 bytes. Functions that read a string from the memory start from the address given by the user and end when they see a null. The resulting string is returned to the user.

Both virtual and physical memory addresses can be used with read and write operations. The third option is to directly read from kernel symbols. When a kernel symbol read is done, the LibVMI manages all potential page boundary related issues, resolves the kernel symbol, translates the symbol to a physical address and performs the actual read operation. For Linux guests, the System.map file is used in the address translation. Kernel symbol read functions are a powerful tool as they allow the user to directly reference kernel symbols such as *PsActiveProcessHead* and *PsLoadedModuleList*.

Memory write functions work just like the read functions. The same memory chunk sizes are also available as write options. The user can choose to write 8, 16, 32, 64 bits, an address (4 or 8 bytes) or an arbitrary number of bytes (lengt is specified in a parameter) to the memory. The same address type options (virtual, physical and kernel symbol) are available for write functions and the function names follow the same pattern as the read functions. The function *vmi_write_addr_ksym* would thus write the address provided by the user to the address to which the chosen kernel symbol refers.

LibVMI can be set to monitor for events occurring from the monitored virtual machine. In total, LibVMI has 10 different event type. The most notable event types are memory, register, single step and interrupt events. The process of making LibVMI listen to the desired events has few steps. First the user must create an event structure. Then the user needs to register that event using *vmi_register_event* function. After the event has been registered the vmi instance is set to check for events that match the registered events using the *vmi_events_listen* function. The function will listen to events until it finds one or a timeout has been occurred. If a matching event is found, the event is passed to its call-back function with some additional information that helps interpreting the event.

Whenever an event is triggered, the default and designed behaviour is to also stop the virtual machine. Pausing the VM preserves the state of the VM and allows consistent introspection of memory and registers. If the VM would not be paused some information might be lost as the register values would already have changed. For example, if we use the contents of the CR3 register as identifier of which process is currently in execution and a memory write event gets triggered. When the virtual VPU gets paused at as the event occurs, it is possible to look at the CR3 register and see which process caused the event. If the VCPU is not changed the contents of the CR3 register could already have been changed and It could be impossible to determine which process caused the modification.

There are other pitfalls with event usage. It is possible to cause conditions that crash, corrupt, or indefinitely suspend the monitored virtual machine. For example, if the user program crashes while there are events unacknowledged, the VM can remain in a paused state. The VCPUs would remain paused as the events have not been acknowledged nor has vmi_destroy been called.

Register events are used to monitor for events in the registers of the virtual CPU such as the IDTR or CR3. The event is constructed by stating which register is going to be monitored and what type of access causes the event to match. In the Python wrapper the register event class constructor has six parameters – register, in_access, callback, equal, slat_id and data. The register parameter states which register is going to be monitored. The parameter should be one of the X86Reg enums defined by the LibVMI. In_access parameter states the access type. It is indicaded by RegAccess enums (possible values: INVALID, N, R, W and RW). Parameter callback is a pointer to the call-back function of the event. Parameters equal and slat_id are defaulted to 0. The equal value could, for example be used to track whenever a specific process enters execution. One example of a correctly defined register event is:

```
register_event = RegEvent(X86Reg.CR3, RegAccess.W, cr3_callback)
```

Memory events are used to monitor for memory access on specified regions. It follows the same general form as register events, but with few different parameters. In the Python wrapper, different memory access types (R, W, X, RW, RX, RWX and so on) are defined by MemAccess enumerations in the same way register access is defined by the RegAccess enums. The constructor has a parameter gfn, which indicates the guest frame number where the trap is located.

When dealing with memory events, the user needs to pay attention to page permissions. For example, if a page frame is trapped for write access, the write permission for that page is revoked. Any write access to that page will cause a Page Illegal Write exception to occur which is then trapped to the hypervisor. When an event is received, the page permissions need to be modified so that the execution may complete.

Single step events cause the targeted virtual CPU to perform single stepping. This means that for each instruction, the VM is paused and the execution is transferred to the hypervisor, where the call-back function written by the user will execute. Single stepping is useful for debugging purposes, but it causes major performance overhead.

The performance overhead that comes from trapping various events is related to how frequently the traps occur. Obviously single stepping causes the largest performance overhead as a VMEXIT occurs on all instructions. When modifying register access the performance overhead depends on which register is being monitored. If we set up a system to monitor changes in the CR3 register, an event would be triggered on every context switch (the process in execution is changed). When monitoring write changes to the CR2 register, we would get an event whenever a page fault occurs. As the performance overhead depends on what is being trapped, the designer of a VMI solution needs to carefully figure out whether the performance overhead caused by the additional traps is acceptable for the purposes of the solution.

7 Malware analysis with virtual machine introspection

Virtual Machine Introspection lends itself well for malware analysis. Malware analysis platforms that are based on virtual machine introspection have full control and visibility over the monitored platform. Those solutions are also harder for malware to detect as there are no in-VM components that could easily be detected. The discovery of VMI-based monitoring solutions is based on measuring execution times and determining whether any introspection is being done on the target (timing based detection). As discussed in earlier chapters, the main challenge for VMI-based solutions is bridging the semantic gap, which essentially means interpreting the low-level hardware state gained from the hypervisor.

There are three main phases for any comprehensive malware analysis. Those phases are behavioural, code and memory analysis. The analysis can start from any one of these phases and cycle through them without any particular order. This chapter takes a look at how these phases can be performed using VMI. [25]

7.1 Behavioural analysis

Behavioural analysis phase examines how the malware sample interacts with its environment. The goal is to build a view of the capabilities and goals of the malware sample. To build that view, all interactions with the malware's operating environment (system calls, file system interactions, Windows registry modifications and network connections) are recorded and analysed. [25]

Traditional in-VM tools that are used in behavioural analysis include the Windows Sysinternals suite [26] for monitoring and logging the registry and file system accesses in Windows environments and Wireshark [27] to monitor for network traffic.

For behavioural analysis to work properly, it is crucial that the sample does not become aware of the monitoring system as it can then change its behaviour to mislead or hinder the analysis [1]. For in-VM tools, simply the presence of the monitoring tool can lead to the malware modifying its behaviour. For VMI-based monitoring platforms, limiting the caused performance overhead is crucial as it can lead to exposing the presence of the monitoring platform. In the context of malware analysis, the performance overhead should be viewed mainly as a stealth issue.

Many of the VMI-based applications presented in the precious chapter are intended to be used for malware analysis (Nitro, Ether, DRAKVUF). More specifically, they are designed to capture the behaviour of the malware sample through monitoring and logging the usage of system calls. Regardless of the method system calls are trapped, the solutions need to provide few information fields to allow meaningful analysis. For each logged system call at least the calling process and system call number needs to be logged (many malware classifiers are interested in sequences of system calls rather than system calls with specific parameters [28]). These tasks can be done by reading the values from few hardware registers. The hardware register values can then be converted into human readable form using correct mappings (for example system call number to system call name from system call table).

To get a more detailed image of the execution of the malware sample, additional information such as system call parameters can be collected. The system call parameters can give additional insight on potential artifacts that could be used as Indicators of Compromise (IoC). In fact, extracting IoCs is one of the main goals for deeper malware analysis. If such IoCs can be obtained, instances of the studied malware can be discovered on other devices by looking for the indicators. For example, McAfee's Enterprise Security Manager supports IoCs such as file names and paths, file hashes and Windows registry values [29].

DRAKVUF extracts full system call trace of the analysis environment. Each log row that represents a system call contains timestamps, system call number and name as well as parameters used to invoke said system call. On returning from system calls DRAKVUF extracts the returning status of the system call.

DRAKVUF's regmon plugin covers Windows registry modifications. Being able to monitor Windows registry is an important trait as malware often uses it to perform various tasks. For example, the run keys are commonly used to maintain persistence by setting the malicious code to be launched every time the user logs in [30]. Such Windows registry artefacts are common IoCs that come from behavioural analysis.

7.2 Code analysis

Code analysis is the phase, where the analyst attempts to gain an understanding of the malware sample by looking at its code. The malware binary is first put through a disassembler, which converts the machine instructions from their binary form into assembly code. A decompiler can then be used to try to recreate the original source code of the sample. The final task is to step through interesting code sections using a debugger. The debugger allows the analyst to execute the malware one instruction at a time. Many malware use modules that attempt to find out if the malware is being analysed and change behaviour accordingly [1]. For example, IDA PRO is a widely used disassembler. [25]

Even though disassembling and decompiling the execution binaries of malware samples are important tasks in malware analysis, they will not be covered in this thesis as they do not need a VMI-based approach (malware samples are not executed). Debuggers on the other hand can be done by leveraging VMI.

Debugging from the hypervisor is beneficial due to increased stealth and visibility. For malware analysis purposes operating system API's can be problematic as they were not designed for dealing with malicious code. As a result they lack the desired stealth capabilities (malware can detect that it is being debugged by a user-mode debugger by calling Windows function IsDebuggerPresent [31]). There might also be kernel security mechanisms present that limit the debugger's view of the system [24]. Doing debugging from the hypervisor also enables the analyst to modify register values.

Existing VMI-based debugging platforms include xendbg [32], pyvmidbg [24] and rVMI [33].

7.3 Memory analysis

Memory analysis is the phase, where the focus of analysis turns to identifying malicious artefacts in the memory of the analysis environment [25]. Such artefacts include rootkits that attempt to remain hidden. Tools that are used in memory analysis include the Volatility framework. In fact, Volatility can be used directly for memory analysis as there is a Volatility address space plugin, which allows Volatility to inspect the memory of virtual machines through LibVMI.

Virtual Machine Introspection suits well for memory analysis. Normally memory analysis is done against memory snapshots, but with VMI the analysis can be done on live machines. This enables setups, where Volatility is run immediately after an interesting event has occurred. One such setup is setting up a high-interaction honeypot and configuring LibVMI to call specific Volatility plugins once changes occur in the memory of the monitored VM as done in [34].

DRAKVUF's poolmon plugin indirectly operates within memory analysis section as it keeps track of pool allocations within the Windows heap. The purpose for the plugin is to keep track of all pool allocations so that we know the location of all allocated pool blocks. The authors of DRAKVUF argue that the root cause for DKOM (Direct Kernel Object Manipulation) attacks is that the exact location of pool blocks becomes unknown within the kernel's heap and by keeping track of all pool allocations, the exact location can never be unknown thus removing the need for pooltag scanning. [20]

The VMI-based solutions discussed in Chapter 5 do not directly deal with memory analysis phase. The solutions are mainly used for behavioural analysis. Apart from few DRAKVUF plugins such as file carving and process memory dumping, the main memory analysis capability currently comes from integrating Volatility with LibVMI. LibVMI handles the complex task of walking through guest page tables and address translation and Volatility bridges the semantic gap.

7.4 DRAKVUF as a malware analysis platform

DRAKVUF is a feature rich malware analysis platform intended to be used in the behavioural analysis phase. As mentioned in Section 7.1, the goal of behavioural malware analysis is to understand how the sample interacts with its operating environment. To do that, one needs to have various sensors in place that can be used to record and monitor the sample's behaviour.

In DRAKVUF, the sensors are implemented as plugins. Overall, DRAKVUF has 25 different plugins. Each plugin is developed to monitor for specific behaviour. Combining these plugins increases the visibility of the sample's behaviour.

MITRE's ATT&CK framework [30] has a collection of adversary techniques and tactics used to achieve specific milestones such as persistence, privilege escalation, and impact. Being able to map the techniques and tactics used by the malware tremendously increases understanding of the malware sample.

The following list contains a list of the most prevalent data sources in Mitre's ATTCK framework [30]. The data sources are data sources, which can be used to identify malicious behaviour. The remainder of this section is focused on analysing how DRAKVUF can be used to tap into those data sources.

DS1 Process monitoring.

DS2 File system monitoring.

DS3 API call monitoring

DS4 Network usage monitoring.

DS5 Windows registry usage monitoring.

System calls are user-level programs way for requesting operating system services such as virtual memory management, thread management, file system management and network connection management. Due to their key involvement in process's execution, system call analysis is a popular task in behavioural malware analysis. System call traces can be used to classify malicious processes and identify malicious behaviour [28]. For malware analysis platform, extracting the system call trace of the executed malware sample is key functionality. DRAKVUF allows the user to monitor all system calls or manually select a subset of system calls for monitoring. This flexibility allows the analyst to remove uninformative system calls from the execution trace and thus reduce noise.

Many of the data sources mentioned in the list of key data sources can be covered by monitoring system call usage. The activities of a process can be explained by looking at the system calls and their parameters. The parameters alone may not be enough to give a detailed description of the event. For example, many system call parameters are handles (file, port, key, process, etc). Gaining more insight on the resources the handle represents would require accessing the handle table.

By monitoring system calls, one can gain insight on the activities of a process. As DRAKVUF extracts the name and path of the caller process, one can easily map issued system calls to their calling process and thus DRAKVUF satisfies DS1 (process monitoring). System call monitoring alone gives some insight on DS2, DS4, and DS5 as there are system calls that handle related behaviour. But as many parameters are handles, additional effort is required to get a clearer view of the behaviour of the sample.

To tap into DS2, one can use DRAKVUF plugins filetracer and filedelete. The plugin filedelete monitors file modifications and file deletes and can be used to extract removed or modified files. Filetracker monitors for file system access and extracts the file names of the accessed files. The filetracer plugin traps seven file system usage system calls such as CreateFile, WriteFile, and ReadFile.

To gain information from DS3, one can use DRAKVUF plugin apimon. And DS4 can be accessed by using socketmon, which monitors TCP and UDP sockets for Windows guests. By monitoring network connections with DRAKVUF, one can analyse the network connections per process instead of per host (which is the case in network-based).

Windows registry (DS5) can be monitored using regmon plugin. It monitores 14 Windows registry related system calls and parses the key handle parameters of the system calls. As a result, the log rows contain the process responsible for the system call as well as the accessed registry key and the new value (if applicable).

In addition to tapping to different data sources, DRAKVUF plugin debugmon can be used for monitoring hardware debugs. Malware sometimes has self-debugging capabilities, which prevent the analyser to attach other debuggers to it. If malware attempts to do some sort of self-debugging, the plugin can be used to monitor that.

DRAKVUF does not incorporate any features for malicious process identification. It is solely focused on developing sensors, which collect data that can then be passed to other tools for classification or analysis. To facilitate better interoperability with other malware analysis tools, logs from DRAKVUF can be output in csv and json format.

8 Estimating the performance overhead of Virtual Machine Introspection

8.1 Detecting the monitoring solution from the VM

One of the main arguments for a VMI based security solution is its ability to remain hidden from the analysis target. There is considerable design effort in many VMI-based solutions to keep the guest oblivious to the presence of the monitoring system. This means that there should be nothing inside the monitored systems that directly or indirectly exposes the presence of the monitoring solution. Even the presence of simple auto-run scripts that launch the malware sample can be used as indication of the monitoring solution. [35]

Uitto et. al [22] present and categorize different detection vectors that malware can use to detect if the environment they are running in an environment where they are being monitored. The detection vectors are divided into four categories - temporal, operational, hardware and environment. Detection vectors inside temporal category rely on timing information to expose the monitoring environment. Operational detection vectors rely on identifying which operations are allowed in their environment. Legislation can pose some limitations on which operations can be allowed to be performed in the monitoring environment (for example, further malware propagation cannot usually be allowed). Hardware category focuses on finding system anomalies. These detection vectors include drivers that the monitoring environment creates and some VM dependent hardware models. The environment category relies on the discovery of in-VM artefacts such as running processes and installed programs.

Out of the four detection vector categories (temporal, operational, hardware and environment) in [22], the local temporal category is of particular interest. The detection vectors in the operational category are ignored in the following tests due to the limitations legislation poses on allowing all operations. Hardware category is also ignored as we are focused on identifying the monitoring environment, not that the environment is being virtualized. Finally, the detection vectors in the environment category are also dropped as we assume there are no in-VM components present in the system (out-of-VM category). The local temporal detection vector will be the focus of Sections 8.4 and 8.5 due to the performance overhead caused by the additional VMEXITs.

Solutions such as Ether and Nitro do not make any modifications to the monitored guest OS [4][2]. Transferring execution to the hypervisor relies on events that can natively be trapped. Those solutions do however modify the hardware resources so that events that would not normally cause a VMEXIT now cause one (see Section 4.3 on trapping system calls). DRAKVUF on the other hand inserts breakpoints directly to OS code [20]. These modifications are kept hidden by relying on access rights on the Extended Page Table. [35]

In addition to discovering any modifications to the guest OS or any in-VM agents, the monitoring solution could be detected using timing analysis. The principal idea behind timing analysis is to observe how long does it take for a certain piece of code to execute and see if it matches the expected time value. The purpose is to observe the overhead caused by the additional VMEXITs and thus infer that there is an introspecting hypervisor present. [35]

8.2 Performance degradation caused by Virtual Machine Introspection

The VMEXITs are the key source for performance degradation in virtualized environments. A VMEXIT occurs whenever an event occurs which needs to be handled by the hypervisor. For example, the hypervisor maintains shadow page tables, which are used to tell how guest physical addresses are mapped to host physical addresses. Whenever a page fault occurs within the virtualized system, the hypervisor must step in to update the mapping. In environments where the hypervisor does introspection, these VMEXITs are more frequent and the time spent executing hypervisor's code is longer. [35]

To facilitate faster world switches, the virtualization extensions maintain a special data structure called VM Control structure. It consists of six logical groups that handle hypervisor operations and state transitions between the guest and the hypervisor. When a VMEXIT occurs, the processor first records the cause of the VMEXIT. Then it must save the current processor state (control registers, debug registers, RIP and so on) to the guest state area of the structure. Then it stores MSRs in the MSR-store area. The last step is to load the host processor state from the host state area and the host MSRs from MSR-load area. After this world switch has been made, the hypervisor can perform its duties and transfer execution back to the VM (the same steps in reverse order). The cost for a world switch is of the order of hundreds or thousands of cycles per transition. [35][36]

As is evident, there is some performance overhead inherently present in virtualized environment which arises from the additional system management complexity and thus the need for VMEXITs. The task of detecting the monitoring solution is not the same as detecting the virtualized environment. As virtual machines are widely deployed in today's business environments, there is financial incentive for malware to continue running normally even in virtualized environments. [35]

There are few timers and timing methods available – the *timestamp counter (TSC)* and *high precision event timer (HPET)*. The TSC is a high precision timer which can measure individual processor clock cycles. It is typically used as a timing source for timing-based side-channels. The hypervisor can intercept requests for the TSC values and adjust the value so that it does not expose the monitoring system [4]. The HPET timer is less accurate, but harder to tamper with as it is commonly used as a source of synchronization for multimedia streams. If the HPET would be manipulated, it would cause issues with user interface animations and video playback. A third timer approach would be to execute code segments with known timing properties in parallel with operations of interest. The timer is used by counting the amount of loops it runs while the operation of interest runs. For this timer to work, the system needs to be multi-threaded. [35]

The magnitude of performance overhead caused by the monitoring solution is thus dependant on two factors. The first factor is the number (f) of additional VMEXITs caused by the monitoring solution and the second factor is the amount of instructions (M) required to perform the task once the VMEXIT has occurred. Let us call these factors *trapping frequency* and *average task magnitude* respectively. The number of additional VM-exists is dependant on the trapped events. The more events are being trapped; the greater visibility is gained on the activities of the guest while simultaneously introducing more performance overhead.

The second factor describes what the hypervisor does with the event once it receives the event. Accessing the guest's memory to construct the higher-level information about the events requires more effort than logging few register values.

8.3 Motivation and research goals

When looking at the performance overhead caused by virtual machine introspection, one can take two different analysis approaches. The first approach would be to look at performance overhead solely as a stealth issue. Such an approach is meaningful particularly when dealing with solutions that are intended to be used only as malware analysis platforms. The decreased performance values and increased wait times are only important if they can be used to detect the monitoring platform.

The second approach would be to look at the performance overhead as a usability issue. If the performance of the virtual machine degrades so that it can no longer perform its intended task within acceptable time, the overhead is too large. This approach lends well to scenarios, where the VMI-solution is deployed as a continuous service such as an IDS or a SIEM log-source. Potential deployment scenarios could be securing and monitoring legacy systems and virtualized servers.

Monitoring invoked system calls is a common task in behavioural malware analysis. The system call trace of the system is collected and studied for indicators of malicious behaviour. According to to Canzanese et. al. [28], system call frequency alone is not expressive enough to be used as an indicator. Instead, a sliding window approach is required. One frame of the sliding window is called an n-gram. The n-gram represents n consecutive items from the sample, which in this case is the system call trace. Hence, a 3-gram would represent three consecutive system calls caught by the monitoring solution.

Depending on the level of trapping, the observed n-grams can be the true sequences of system calls or a portion of the system calls can be omitted from the system call trace. Assume our system uses five different system (numbers 1,2,3,4,5) calls to perform all necessary tasks and we use 3-grams to represent the execution trace. Assuming a program invokes system calls (1,2,4,3,5,2,4). Our monitoring solution would then observe the following 3-grams (1,2,4), (2,4,3), (4,3,5), (3,5,2), and (5,2,4). If our monitoring solution were to trap system calls 1,3,4, and 5, but not 2, the system call trace observed by the solution would be (1,4,3), (4,3,5), (3,5,3). In this example, only one of the actual 3-grams is present in the new execution trace.

For visibility point of view, we would ideally want to trap all system calls so that our view of the sample's execution does not get altered. For malware analysis focused on understanding the behaviour of a malware sample, gaining the full execution trace is desirable as it shows how the sample interacts with its environment. If the goal is to classify programs as malicious/benign, choosing a subset of all system calls for monitoring may be desirable as it reduces the model complexity and the performance overhead of the monitoring solution. The chosen subset needs to be expressive enough so that it can still correctly classify processes. Choosing such subset is not within the scope of this thesis.

If we choose to use n-grams to represent execution traces of programs, the number of potential unique n-grams increases quickly. Let us denote the number of monitored system calls by *k*. Using this denotation, the maximum number of unique n-grams is k^n. Because of this relationship between n and the maximum number of unique n-grams, the chosen n cannot be high. Increasing n also increases model complexity and may lead to overfitting. In [28] the value of n was selected to be 3.

The objective of the test scenarios presented in Section 8.4 is to answer to the following test goals:

TG1 Determine the feasibility of trapping all system calls

TG2 Estimate the cost of trapping an event

TG3 Determine a reasonable frequency for trapping events so that the performance overhead is within acceptable boundaries

TG4 Evaluate the performance of important DRAKVUF plugins (system call monitoring, file delete capturing, and pool allocation monitoring)

8.4 Test scenarios

Test environment

Dell Latitude E7440 laptop with Intel i5-4300U (2.49 GHz) dual core processor and 8GB of RAM running Xen hypervisor. The Dom0 is running Ubuntu 18.04.1 and it is designated one processor core and 4GB of RAM. The guest is a Windows 7 SP 1 Professional, with 2GB of RAM and one processor core.

DRAKVUF is run from the dom0 with various plugin and parameter combinations. These parameters are explained in greater detail per each test.

8.4.1 Test 1: Calculating Fast Fourier Transforms

The first test is performed by using a stress testing tool Prime95 inside the guest VM. The tool was used to run Fast Fourier Transformations and then measure the time it takes for the calculations to run. The measurement is repeated for 27 different sizes ranging from 2048K to 8192K. As the FFT size increases, so does the execution time.

This tool was selected to evaluate the performance overhead of CPU intensive task. It differs from the setup in Test 3 as it does not issue as many system calls as browser usage. Tests 1 and 3 should show the difficulty of analysing the performance overhead caused by virtual machine introspection as different types of tasks have very different performance overhead.

The measurements are carried out two times. Once with no monitoring elements and once using DRAKVUF to monitor for system calls. The time difference between the two cases should give some insight on the magnitude of the performance overhead caused by monitoring system calls from the hypervisor in a CPU intensive task.

The measurements can then be used to answer to Test Goals TG1 and TG3. To estimate the cost of single trapped event, one would need to have data on how many events were trapped at all FFT lengths. The test does not provide this data.

8.4.2 Test 2: Novabench desktop benchmarking tool

The second test is designed to estimate the performance overhead of monitoring system system calls. The test is carried out by first running Novabench without DRAKVUF monitoring, which gives a baseline of system performance. After the baseline is formed, DRAKVUF is switched on and the benchmarking tool is ran again.

Novabench is a benchmarking tool that is used to give hardware components performance scores. In this case we are interested in the numbers behind the performance scores. In this setup, the Novabench tool gives two different performance values - one for the CPU and another for RAM. The CPU score is calculated from three different measurements: floating point ops, integer ops and hash ops. The RAM score has only one additional attribute – RAM speed.

Novabench was selected because it offers three different attributes to evaluate CPU performance (hash, integer and float point operations). The attributes show whether the performance drop is consistent between all three attributes.

8.4.3 Test 3: System call monitoring with simulated browser usage

This test scenario measures performance overhead in a scenario, where system calls are frequent. The discoveries of this test are directly applicable in determining, whether it is feasible to capture all system calls. Gain insight on Test Goal TG2 and TG3, additional calculations are performed on the measurements.

Simulated browser usage is chosen as the payload to be run inside the monitored VM, as they take heavy use of operating system features (filesystem, threads, sockets, etc.) and as a result they issue a large number of system calls. To keep the load similar between all measurements, a Python script is used to open three websites in a browser.

As part of this test, the performance overhead of the system call monitoring is measured when trapping only a subset of all system calls. The system call subsets are chosen so that the entire set constitutes p percent of the entire system call volume (of the selected load).

In order to be able to determine such subsets, one first needs to find out the frequency of all system calls for the selected load. The system call frequencies are determined by trapping and logging all issued system calls during a 60-second window while running the Python script. After the 60-second execution trace has been gathered, a Python library Pandas is used to parse the logs and determine the system call subsets. The system calls belonging to each subset are selected randomly for each percentage level. After a subset has been determined, the combined system call volume is checked to ensure that it does not represent too large portion of all system calls.

The metric which indicates how many system calls are trapped was chosen to be the volume of system calls rather than the number of unique system calls. Trapping p percent of all system calls gives a sense of how much performance overhead is caused by intercepting p percent of system calls. If we would have selected x of available system call numbers, the measurements would not give any meaningful insight on performance overhead as the frequency of system calls varies a lot between system calls. A set containing 100 unique system calls could have been trapped much more frequently than another set containing 200 unique system calls if the system calls in the former set occur more frequently.

Each measurement goes as follows: First a bash script is launched from dom0, which launches the DRAKVUF's system call monitoring with a list telling which system calls should be trapped. Then a second script is launched inside the domU, which loads two Python libraries (time and webbrowser) and then opens Google's website on a new browser. Once the website has fully finished loading, the bash script in dom0 is interrupted using a keyboard interrupt and the execution time measured form dom0 is given. These steps are repeated three times for each measurement level and the average execution time is calculated.

As the execution time is measured from dom0, the Python script in domU is launched manually and the measurement is halted manually using a keyboard interrupt, there is always some error present in the values. However, as the execution time of the task is fairly long, the human error introduced by the test setup does not contribute too much to the end results.

As DRAKVUF's system call monitoring plugin works by injecting breakpoint instructions to the monitored system calls, the timing differences between different levels are expected to be fairly linear. In Nitro's and Ether's case where system calls are trapped as described in Chaper 4, the behaviour might be non-linear.

Choosing the system call subset

The Algorithm 1 shows the method how the system call subset is chosen. The algorithm has three parameters: A,t and p. Parameter A is a structure which is built from the system call trace. It contains the names of all observed unique system calls and their corresponding appearance frequency in the system call trace. Parameter t contains the total number of system calls in the system call trace. Finally, parameter $p \in (0, 1)$ tells how many percent of all system calls should the subset cover. As output, the algorithm gives the names of the system calls, which form the desired subset. When compared to the system call trace, the subset represents minimum of $p \cdot 100\%$ and maximum of $(p + 0.01) \cdot 100\%$ of all system calls.

It is worth noting, that the resulting system call subset covers only $p \cdot 100\%$ of all system call volume for the extracted system call trace. When the subset is applied again, the exact percentage is likely to differ. For the same load, the percentage of captured system calls is fairly close to p as the system call trace is similar. If the load changes, the subset no longer represents $p \cdot 100\%$ of all issued system calls (for example, switching the load from browser usage to image manipulation).

8.4.4 Test 4: File carving overhead measurement

This test focuses on measuring the overhead of DRAKVUF's filedelete plugin, which carves all removed and modified files from memory. Unlike other measurements, this measurement also covers the disk space requirements of running this plugin.

File carving is an important capability in malware analysis and forensics. The impor-

Algorithm 1 Selecting the system call subset.

```
1: procedure SYSTEM_CALL_SUBSET($A$,$t$,$p$)
2:     subset ← empty list
3:     $c \leftarrow 0$
4:     while $\frac{c}{t} < p$ do
5:         $s \leftarrow$ random system call from $A$
6:         $f \leftarrow$ frequency of $s$
7:         $c \leftarrow c + f$
8:         append $s$ to subset
9:         if $\frac{c}{t} > p + 0.01$ then
10:            remove $s$ from subset
11:            $c \leftarrow c - f$
12:    Return subset
```

tance of the capability stems from the fact that malware makes heavy use of temporary files. For example, Trojan Downloaders often download the payload over the network and save it in temporary file. Once it is no longer used, the file will be removed. Being able to capture these files can help identify the malware, understand its behaviour and mitigate future campaigns of the same malware variant. [37]

The performance overhead is measured in the same way as when measuring the performance overhead of system call monitoring. A bash script is used to launch DRAKVUF with filedelete plugin and simultaneously measure the execution time. Once DRAKVUF has been launched a script is launched inside the monitored guest which opens three different websites and waits for 10 seconds between each website. Once all websites have finished loading, the bash script is interrupted, and the execution time is given. Finally, the size of the folder containing the dumped files and their metadata is checked as well as the file count inside the folder. The measurement is repeated three times and averages for all values are calculated.

For this test, browser usage is an excellent load. Browsers use lots of temporary files such as cookies and as a result, we should see some file system activity related to file deletes.

This test scenario no longer gives any insight on Test Goal TG1 as it does not trap all system calls. The answer to Test Goal TG2 should differ from Test 1, 2, and 3 as the task performed after the event has been trapped is heavier.

8.4.5 Test 5: Pool allocation monitoring overhead

The final test scenario covers the performance overhead that results from monitoring only pool allocations in the kernel heap. Pool allocation monitoring is done in order to mitigate Direct Kernel Object Manipulation attacks (for example hiding malicious processes).

The test functions the same way as the previous test. A bash script is used to launch DRAKVUF and measure the execution time and a second script is launched to open three different websites in a browser. Once all sites have finished loading, DRAKVUF is interrupted and the execution time is reported. The test suffers from similar accuracy issues as Tests 3 and 4.

This test scenario is not as customisable as Test 3, where we can select a set of system calls for trapping. DRAKVUF's pool monitoring plugin injects breakpoint instructions to kernel functions responsible for heap allocations (ExAllocatePoolWithTag). As a result, all calls to the functions are trapped. Each trapped event represents activity that the plugin is designed to monitor and hence there is no way to lessen the number of traps caused by the plugin.

8.5 Results

8.5.1 Test 1 results: FFT overhead

When looking at the graph showing the execution times of calculating FFT of various lengths, there seems to be one data outlier at size 3360K. This outlier will be removed from all following calculations.

The curve which represents execution time without DRAKVUF is constantly below the monitoring curve (apart from the data outlier), as expected. The difference between the two curves is not big, indicating that the performance overhead in CPU intensive calculation is not too high.

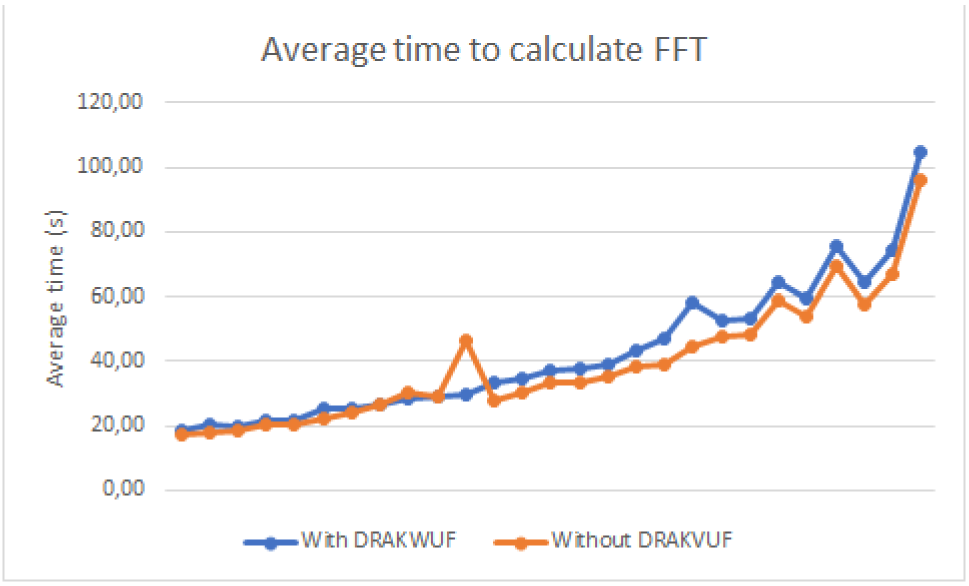


Figure 8.1: Graph showing the average time it takes to calculate Fast Fourier Transform with and without DRAKVUF's system call monitoring plugin. The sizes of the FFT range from 2048K to 8192K.

The difference between the two curves increases as the FFT size increases, which is expected. As FFT size increases, the difficulty of the calculation (or required calculation power) also increases, which is evident from the steadily increasing execution time. Task size increases, it is natural that the time difference between the execution times also increases.

A more meaningful metric here would be the relative time difference $\frac{\Delta(t1,t2)}{t2}$, where $t1$ represents the execution time with DRAKVUF activated and $t2$ represents the execution time without DRAKVUF activated. The relative time difference indicates how much longer does the calculation take with DRAKVUF than without it.

The relative time difference has an average value of 0.11 and its standard deviation is 0.059. During the measurements, the monitoring solution trapped roughly 20000 system calls during a 60-second time frame, which leads to trapping frequency of 666 Hz (as all system calls are trapped on entry and on return). The average relative time difference indicates that the performance overhead of a task, which issues system calls at a rate of 300 system calls per second, leads to performance overhead of 10 %, which should be both manageable and hard to detect.

Absolute performance overhead of a single trapped system call cannot be calculated from this data as it would require knowing how many system calls were performed when calculating the average execution times for each length. The measurement does not provide data that could be used to answer the question.

8.5.2 Test 2 results: Novabench scores analyzed

The Novabench benchmarking tool tells a different kind of story. The CPU score values drop from 173 to 70 when the system call monitoring plugin was activated. The CPU score dropped roughly 60% when DRAKVUF was activated.

Integer, float point, and hash operations suffered a drop of the same magnitude. The amount of float point operations dropped by 50% and the amount of integer and hash operations dropped over 60%.

RAM score on the other hand suffered a much smaller drop. The RAM score dropped just under 25%, even though the RAM speed dropped from 13500MB/s to 4500MB/s, which is over 60% decrease in speed. The slow RAM speed does not directly indicate

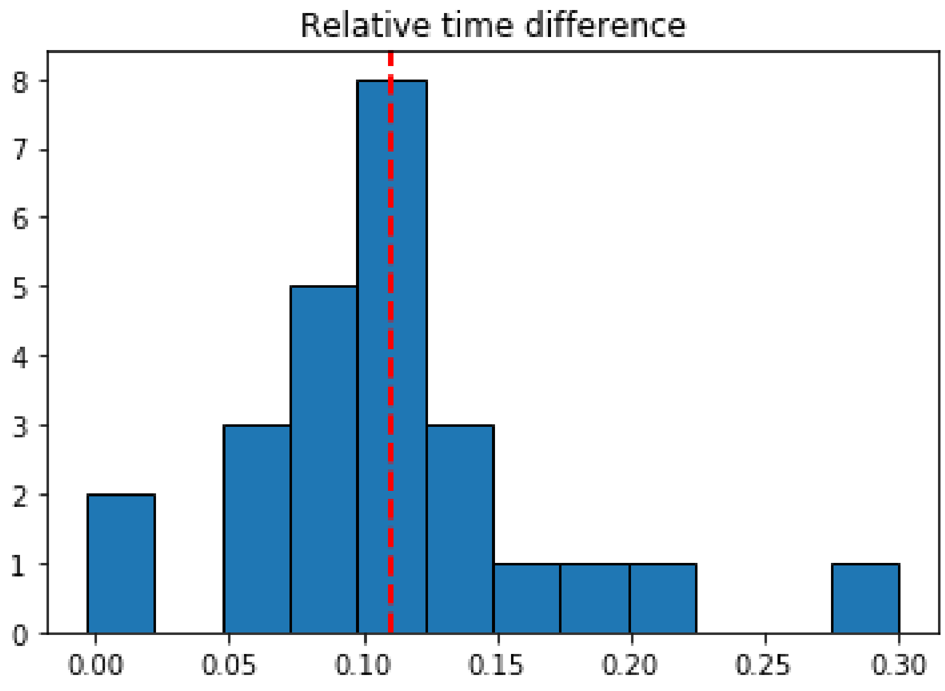


Figure 8.2: Histogram of relative time difference. The red dashed line represents the average relative time differenece 0.11.

that the environment is being monitored but it severely limits usability.

Out of 132000 trapped system calls 60000 were issued by the NovaBenchGUI process. The second most frequent caller process was Windows process explorer.exe with 30000 issued system calls. Among the 25 most frequently issued system calls by the NovaBench process were thread management calls such as YieldExecution, Windows registry related system calls such as QueryKey, OpenKey, and EnumerateKey, memory management calls such as NtAllocateVirtualMemory and NtFreeVirtualMemory, and handle management calls such as NtClose.

8.5.3 Test 3 results: System call monitoring

The third test measures the performance overhead of system call monitoring when p percent of system call volume is trapped. By modifying p, we modify the frequency of trapped events. After measuring the performance overhead that comes from trapping p

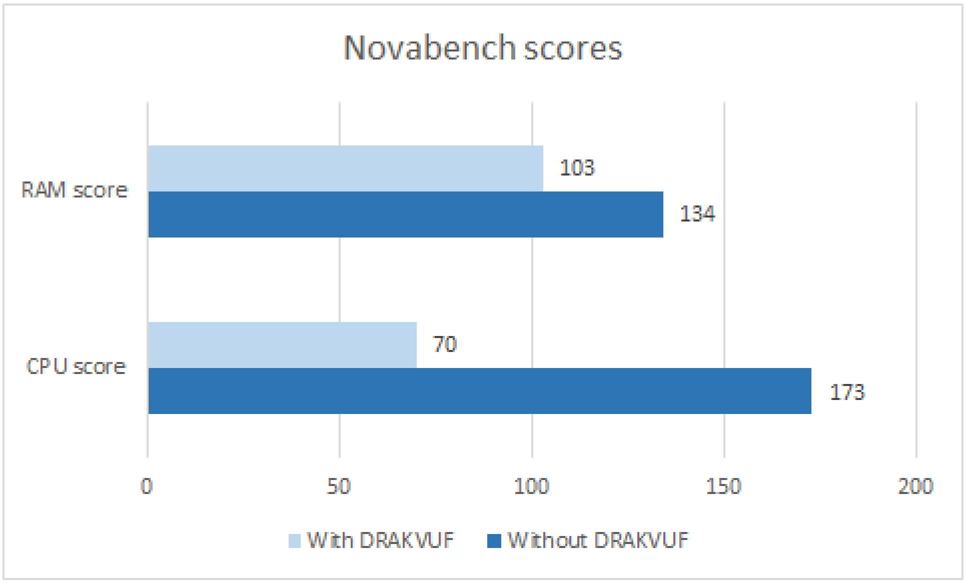


Figure 8.3: A bar chart showing the degradation of CPU and RAM benchmark scores when DRAKVUF's system call monitoring functionality is turned on and off.

percent of all system calls, we can see the effects of trapping frequency on performance overhead and find suitable bounds for system call trapping frequency.

Initial findings

As mentioned in the description of this test scenario, the first task was to gather a 60-second execution trace of the system when running the payload in the VM. The resulting execution trace is then used to find out sets of system calls that cover p percent of all system call traffic.

The initial execution trace consists of over 190000 system calls and their corresponding exits. The highest average event trapping frequency is 6300 Hz (DRAKVUF traps each system call twice) and it is achieved at $p = 100$.

The baseline execution trace has 190 unique system calls. The top 3 most frequently called system calls are NtClose, NtQueryKey and NtQueryValueKey. These three system calls have all been called well over 10000 times during the 60 second window (NtClose got called over 20000 times). NtQueryKey and NtQueryValueKey are system calls related to Windows registry activity. NtClose system call is used to close handles such as files, sockets and threads.

Trapping the three most frequently issued system calls from the initial measurement would already give a minimum trapping frequency of 1600 Hz. As the three system calls all deal with key operating system functionality, much smaller trapping frequencies may not be useful anymore from behavioural analysis perspective as the system call subsets may not be expressive enough to capture the key behaviour characteristics of processes.

Execution time as a function of p

When designing this test scenario, it was estimated that execution time as a function of p would exhibit close to linear behaviour. This is hypothesis is based on the fact that DRAKVUF functions by injecting breakpoints on only those system calls that it wishes to monitor. As a result, issuing system calls that are not monitored do not cause any VMEXITs. The hardware-based trapping methods presented in Chapter 4 would cause VMEXITs even on system calls that are not being monitored.

The linear approximation presented in Figure 8.4 is based on execution time measurements with p getting values from 10 to 100. The linear approximation gave the following function: $y(p) = 0.49p + 13.4$. The coefficient of p^0 should be close to the baseline execution time. The baseline execution time of this test is 5 seconds. The difference between the two values is attributed to poor model performance when $p \to 0$ and the setup time of DRAKVUF. Overall, the linear model of execution time fits fairly well to the measurement data apart from the measurement which traps all system calls. In the linear model, the measurement where p = 100 looks like an outlier.

Let us approximate the execution time as a function of p with a second-degree polynomial $ap^2 + bp + c$. When the data is approximated with a second-degree polynomial, we get function $f(p) = 0.0059p^2 - 0.158p + 26$, which is shown in Figure 8.5. The second-degree polynomial approximates execution time poorly at low values of p ($p \leq 10$).

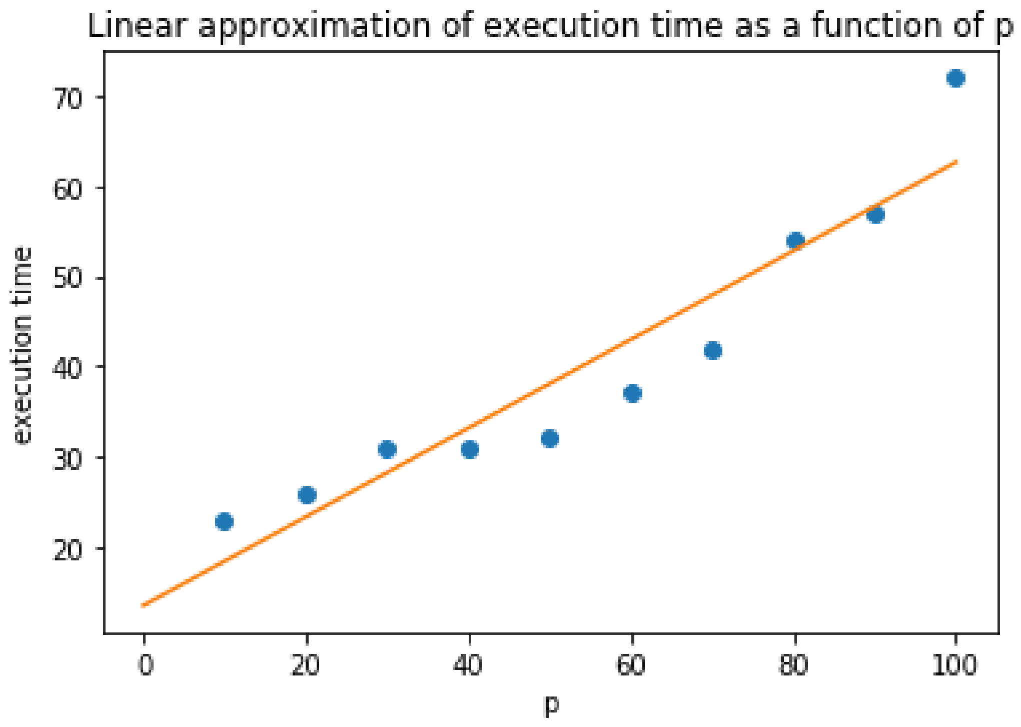


Figure 8.4: Linear approximation of execution time of Test 3 as a function of p. The function of the linear approximation is $y(p) = 0.49p + 13.4$.

However, after $p > 30$, the second-degree model seems to approximate execution time better. Higher degree polynomials are likely to just model the noise in the data.

Trapping frequency as a function of p

In order to try to explain and analyse the performance overhead more closely, the trapping frequency f is presented as a function of p. The trapping frequency is approximated using a linear model (presented in Figure 8.6) and a square root function $a \cdot \sqrt{b \cdot p} + c$ (presented in Figure 8.7). Assuming the average task magnitude M is constant across all system calls, the linear model of trapping frequency supports the linear model of execution time and the square root function model supports the second-degree model of execution time.

The function of the linear approximation is $f(p) = 47.9p + 1315$. The linear model does not capture situation $p \to 0$ well. The trapping frequency at very low values of p is

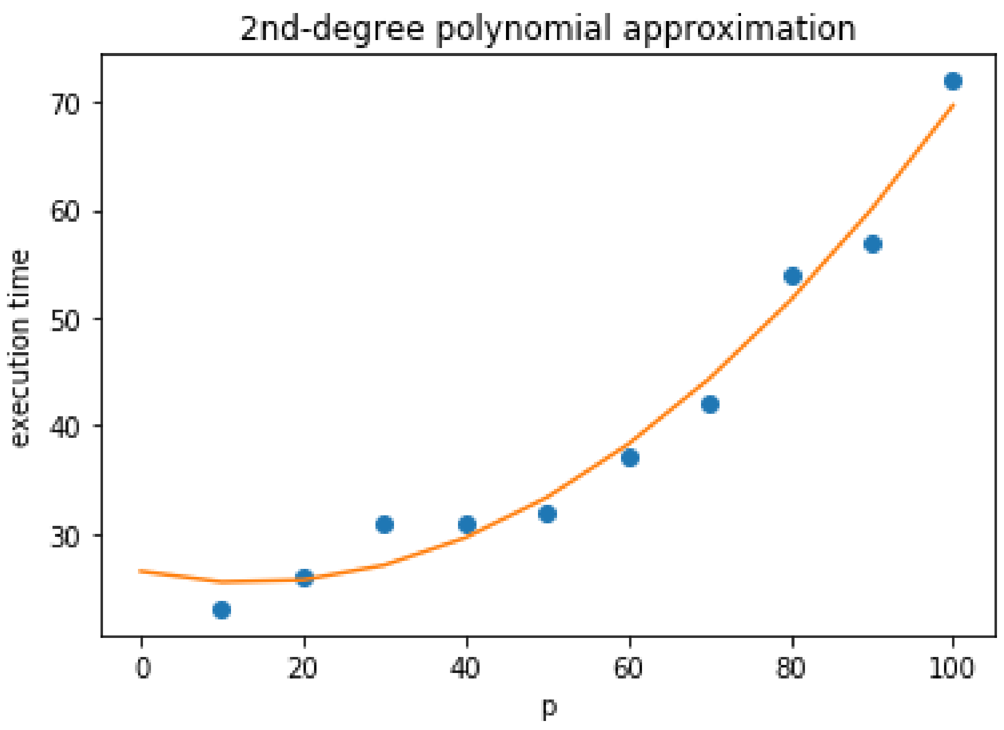


Figure 8.5: Execution time approximated with a second degree polynomial. The model offers poorer performance at lower values of p when compared to the linear model. However, the measurement in which all system calls are trapped fits better into the model.

over 1000 and $f(0) = 1315$.

The square root function of the second model is $f2(p) = 76\sqrt{76p} - 762$. The square root function performs better at low values of p. In fact, $f2(p)$ is equal to zero at $p = 1.32$, which is close to the theoretical root at $p = 0$.

Based on visual observation of the two models and their relation to the data point, neither model seems to be the obvious choice. At $p = 70$ and $p = 90$, the trapping frequency is much lower than expected. Omitting the two values from analysis, the data seems to support the square root model.

The assumption made in the beginning of this subsection was that the average task magnitude is constant across all system calls. This assumption is quite large and it should be analysed whether it holds. The subset of system calls, which is selected for each mea-

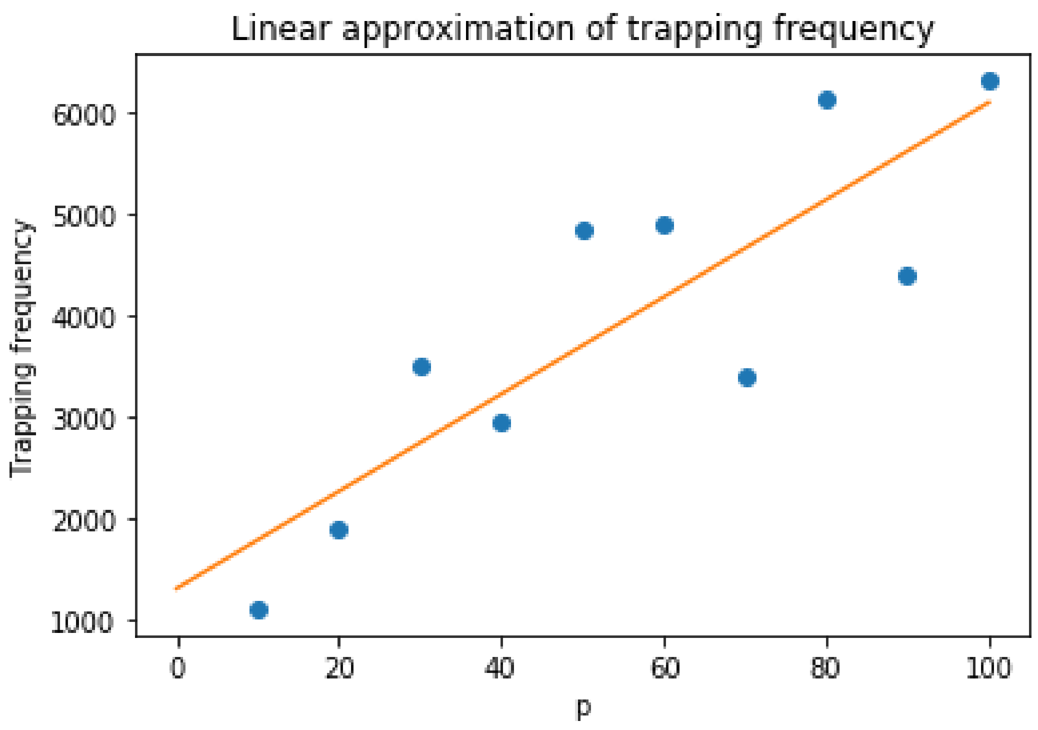


Figure 8.6: Linear approximation of trapping frequency. The linear model performs poorly at low values of p.

surement is chosen at random. As a result, the subsets representing p and $p \pm 10$ percent of system calls may be very different (contain different system calls). DRAKVUF finds all parameters for captured system calls. As system calls have different types of parameters (pointers, handles, integers, etc.) as well as different number of parameters (for example, NtDeviceIoControlFile has 10 parameters and NtClose has only 1), it is likely that the average task magnitude is not constant across all system calls. The unexpectedly low trapping frequencies at $p = 70$ and $p = 90$ could well be explained by trapping events with larger task magnitudes.

After some analysis of the results, the data seems to support the square root model of trapping frequency. Its performance at low values of p greatly outperforms the linear model. Even though the linear model of execution time performs better at low values of p, the second degree polynomial seems to fit the data better particularly after $p > 20$.

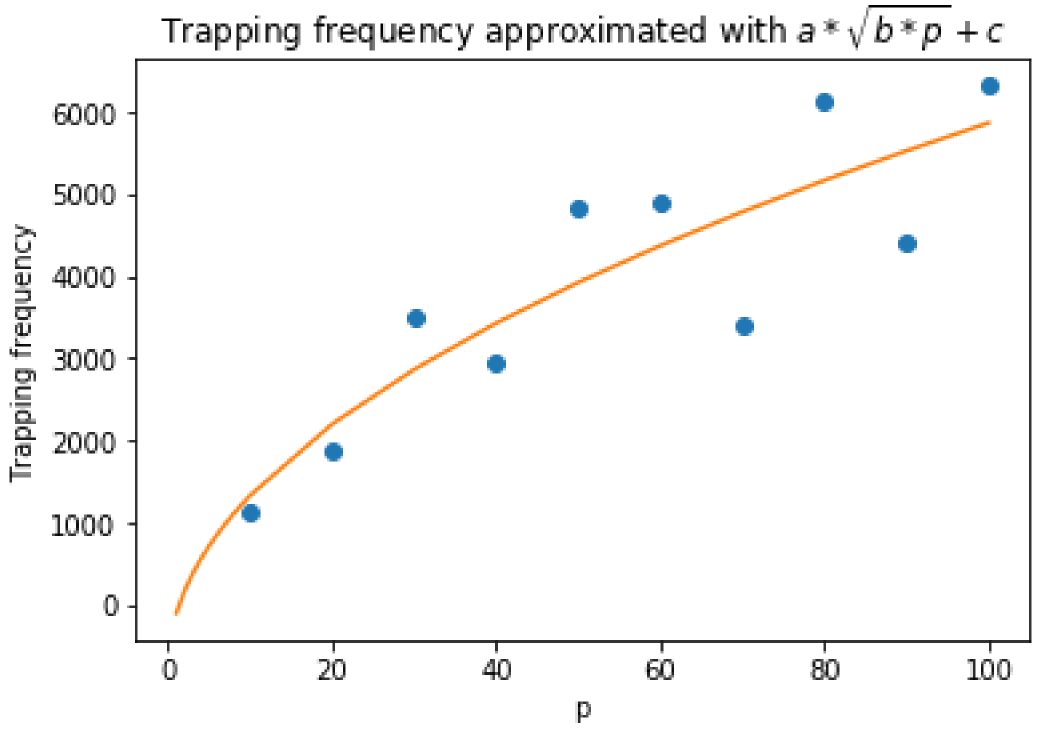


Figure 8.7: Trapping frequency approximated with a square root function $76\sqrt{76p} - 762$. The model seems to perform better than the linear model. The function is drawn for $p \in (1, 100)$.

Comparison to Test 1

The performance overhead of Test 3 differs from Test 1. There are few important differences between the two test cases. The execution time in Test 1 is measured inside the monitoring environment. Similar measurements could be carried out by malicious programs. In Test 3, the execution time is measured from the hypervisor. As a result, set-up time is included in each measurement. This set-up time is small enough so that it does not distort the measurements. The measurements in Test 3 are also ended manually once certain conditions have been achieved (website has finished loading).

The second major difference between the tests is the involvement of Internet as Test 3 requires Internet resources. Internet traffic introduces additional variance to execution time (ping, potential re-transmissions, download speed, etc.). Internet access is not normally available for malware analysis platforms. Honeypots on the other hand require network access.

The third difference between the two test scenarios is the system call trace. In Test 1, the most of the trapped system calls are NtQueryPerformanceCounter. In Test 3, the system call trace consists of 190 unique system calls and the system call trace is more diverse. The system call volume is also much higher in Test 3, which can be seen when comparing trapping frequencies (6300 Hz vs. 666 Hz).

Cost of single trapped event

To calculate the average cost of a single trapped event, one needs to have a repeatable load. The load should be the same when the monitoring is turned on as it is when monitoring is not present. It should start and end in the same condition. In this case the measurement finishes when the website has fully finished loading and starts before launching the Python script.

The average cost of a trapped event is calculated from the measurement, where all system calls are trapped. As a result, we get the average cost of trapping a single event for the chosen load. As trapping some system calls is more expensive than trapping others (for example, different number of parameters), the average cost of trapping a single event will vary if the load changes (or some system calls are no longer trapped).

Using the measurements from case $p = 100$, the average cost of a single trapped event is $\frac{\Delta(t_1,t_0)}{n} = 0.145ms$, where t_1 is the execution time with monitoring turned on, t_0 is the baseline execution time and n is the number of trapped events. The value $0.145ms$ is gained by averaging the calculation for three different measurements.

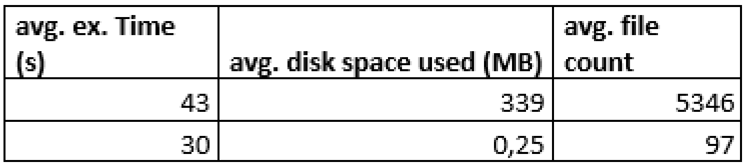
8.5.4 Test 4 results: File carving overhead

The fourth test scenario covers the performance overhead of running DRAKVUF's file extract plugin, which captures all modified and removed files.

The disk usage of the plugin was surprisingly high, which is a result of trapping file writes. As a result, during the 40 second time of execution, the plugin gathered an average of 339 MB of metadata and extracted files. During the three test iterations, the disk usage per iteration reached a maximum value of 486MB.

Upon closer inspection of the extracted events, the majority of them resulted from file write operations. On one iteration only 75 of 4963 trapped events were file deletes. The rest of the events were all file writes. 62 of those 75 file deletes were http cookies. The remaining 13 were various temporary Internet files. Limiting our traps to only file deletes would thus drastically reduce the disk requirements.

The execution overhead, even when trapping file writes, was not too much to handle. The script has a baseline execution time of 30 seconds. It opens three different websites (www.yle.fi, www.google.fi, and www.liiga.fi) and sleeps 10 seconds between all websites (and after the final website). Without DRAKVUF, the last website finishes loading around the time the Python script finally exits. When the filedelete module was activated the execution time increased by an average of 13 seconds.



avg. ex. Time (s)	avg. disk space used (MB)	avg. file count
43	339	5346
30	0,25	97

Figure 8.8: The execution time baseline for this scenario is 30 seconds. Much of the additional execution time for this test scenario attributes to one website taking long to fully finish loading. Over 97% of carved files are the result of file write initiating file carving. Limiting file carving operations only to file deletes would drastically reduce disk requirements. The bottom measurements resulted from rerunning the test without file writes initiating carving.

When the test was rerun without file writes initiating carving, the performance overhead on the system was not noticeable from the baseline execution time at the precision level of the test. As file writes no longer get trapped, the file delete plugin ends up increasing VMEXIT frequency by only 3 Hz.

When only file deletes initiate file carving, malicious programs could trivially circumvent the monitoring solution by first emptying the contents of the file and only after that removing said file. The monitoring solution would not be able to get the original file contents and malware authors could feed misleading data to thwart the analysis.

When file writes initiate file carving, events are trapped at a frequency of 124 events per second. This trapping frequency is by far the smallest of all test scenarios. Despite the lower trapping frequency, the performance overhead is still larger than for example in Test 1 (600 trapped system call events per second). The reason for this difference is that the file carving task performed on all trapped events is very heavy.

When an event which initiates file carving occurs, the monitored VM must remain paused while the event is being handled to keep memory contents consistent. The hypervisor must then locate the file then copy it from the VM's memory to dom0.

The results from this test scenario differ from previous test scenarios as the task magnitude varies more between trapped events. More specifically, the performance overhead of a single event is related to the size of the carved file. A crude approximation of the performance overhead of a single trapped event can be achieved by dividing the difference to the baseline time by the number of trapped events. From these measurements the average cost of a single trapped event is $\frac{13s}{5346} = 2,4ms$.

From usability perspective, disk usage quickly exhausts all realistic storage sizes if file write operations initiate file carving. If file delete operations are the only ones that initiate file carving, the disk usage is at acceptable level. From stealth perspective, the file carving plugin does not expose the presence of the monitoring solution even if file writes initiate file carving.

8.5.5 Test 5 results: Pool allocation monitoring overhead

Pool allocation monitoring offers a lot less possibilities for adjustment than system call monitoring or even file carving. The decision of enabling pool allocation monitoring is binary – either you monitor all pool allocations, or you do not monitor any. The pool allocation monitoring functions by trapping kernel function ExAllocatePoolWithTag. As all pool allocations are done using this function, choosing to log only specific pool tags does not reduce the performance overhead as pool allocations without the specified tag are trapped nonetheless.

The baseline time for this test scenario is 30 seconds. After three repetitions of the measurements an average execution time of 55 seconds was received, which is an increase of over 80% in execution time. During that 55 second, the monitoring solution caused an average of 403562 trapped events, which leads to average trapping frequency of 7332 Hz.

It seems that the task magnitude of pool allocation monitoring is the lightest of all plugins used in the test scenarios. The performance overhead of trapping system calls at nearly the same frequency (6300 trapped events per second, Test 3, $p = 100$), causes much higher performance overhead. The load run at Test 3 is lighter but the execution time is longer. Based on transitivity, the task magnitude of pool allocation is smaller than the task magnitude of file carving.

The performance overhead of a single event can be estimated in the same manner as in Test 4. The difference to base execution time is divided by the number of trapped events. For this task we get, $\frac{25s}{403562} = 0,62 \cdot 10^{-3}ms$.

8.5.6 Summarizing results

After performing the five test cases, it is clear that performance overhead is largely determined by task magnitude. Performing a VM-exit results in close to constant performance overhead. Thus, the differences between the cost of trapping a single event is determined solely by the differences in task magnitude M.

Out of the three different plugins used in the tests, the average task magnitude associated with file carving (filedelete) is the largest and the average task magnitude from pool allocations (poolmon) is the lowest. The average cost of trapping a single event with filedelete plugin is 17 times heavier than trapping a single system call event with syscalls plugin, and the average cost of trapping a single event with poolmon is roughly 230 times lighter than trapping a single event with syscalls plugin. The average costs for trapping a single trapped event are $0.62 \cdot 10^{-3}$, 0.145, and $2,4$ milliseconds for poolmon, syscalls and filedelete.

The average cost of a single trapped system call event from Test 3 is does not conflict with the results from Test 1. The differences between the execution times is attributed to high difference between issued system calls. In Test 3, 380000 events are trapped during a 60-second window. In Test 1, only 40000 events are trapped during the same timeframe. Applying the average cost of a single trapped event ($0.145ms$), the monitoring environment is responsible for $5.8s$ out of the 60 seconds. When the performance overhead in the 60 second window is compared to the time executing guest load, we get $\frac{5.8s}{60s-5.8s} = 0.115$. The average performance overhead from Test 1 is 0.11.

TG1 and TG3 are intertwined with each other. The feasibility of trapping all system calls is dependant on the frequency of system calls. With high system call frequencies (Test 3), it is not feasible to trap all system calls, but with low system call frequencies it is. Formula $P = \frac{t_{mon}}{t_{guest}} \cdot 100\% = \frac{c_{event} \cdot f}{1 - c_{event} \cdot f} \cdot 100\%$ is used to estimate feasible trapping frequency. In the formula, t_{mon}, t_{guest} are the time spent executing monitoring code and the time spent executing guest code, c_{event} is the cost of trapping a single event and f is the trapping frequency. P is the performance overhead percentage that comes form running the monitoring plugin. With $c_{event} = 0.000145s$, $P < 20\%$ when $f < 1150$.

The performance overhead associated with filedelete is within acceptable levels to be deployed as a monitoring solution for virtual environments if file writes do not initiate file carving. The disk requirements can be limited even further, if a cleaner agent is deployed remove uninteresting files after they have been carved or limiting their storage time on the disk. Removing uninteresting files after they have been carved based on some predefined rules essentially means whitelisting some files. Limiting their lifetime on the disk is a standard procedure for logs (for example logs are removed after three weeks from appearance).

From usability perspective the performance overhead of poolmon is too high to be deployed as a monitoring solution. Another argument against deploying poolmon as a monitoring solution is its large log generation. The trapping frequency of poolmon in Test 5 is $7332Hz$. As a result, the monitoring solution would generate 7332 log rows per second, which is over 26 million log rows per hour.

For malware analysis the log row generation of these plugins is not an issue as logs are gathered over a short period of time. From stealth perspective, filedelete and poolmon plugins are unlikely to expose the monitoring environment. If the malware's environment detection is based on issuing a large number of system calls, it may be able to detect the monitoring environment. However, such an approach is not very stealthy. Measurements such as those carried out in Test 1 are more likely, which do not expose the presence of the monitoring environment.

9 Conclusions

When VMI-based tools for malware analysis are compared to in-VM tools (RQ1), the VMI-based tools have few advantages – isolation, full visibility and stealth.

When the monitoring system is deployed outside the monitored environment, malicious code cannot target the security solution directly (isolation trait). In comparison, In-VM tools are vulnerable to tampering as they reside in the same environment as the monitored programs.

As the monitoring solution is not in the same environment as the malicious code, it cannot be detected as easily. In fact, the design of VMI-based tools places emphasis on stealth. The monitored program should not become aware that it is being monitored. To meet this condition, VMI-based tools avoid leaving any detectable artifacts inside the monitored target. The monitored program can attempt to detect the monitored environment by relying on side-channel information such as timing analysis.

As VMI-based tools are placed on the hypervisor, they inherit the full view (and control) of the monitored environment. However, the view of the monitored VM is the hardware state of VM. The hypervisor lacks the operating system specific semantic knowledge, which gives meaning to the observed hardware state.

Knowledge of operating system concepts is fundamental for doing virtual machine introspection. In virtual machine introspection, the hardware state of the virtual machine is inspected and parsed to understand the high-level state of the virtual machine. To be able to construct the high-level state from the hardware state, one needs to apply semantic knowledge. Semantic knowledge can either be based on knowledge of the underlaying hardware architecture (derived) or knowledge of the guest operating system (delivered).

To illustrate the OS-specific nature of semantic knowledge, the practical implementation of processes in Linux and Windows operating systems is discussed in Chapter 2. The intention of the discussion is to show how the two implementations differ from each other. By understanding the differences between implementations of operating system concepts, one can see why bridging the semantic gap using delivered knowledge requires deep knowledge of the guest operating system and its data structures.

To provide an answer to RQ2 and RQ3, VMI-based malware analysis tools Nitro, Ether and DRAKVUF are analysed in Chapter 5. After analysing the tools, DRAKVUF proved to be the most feature-rich and advanced VMI-based malware analysis platform currently available. Chapter 7 focuses on explaining how DRAKVUF can be used for malware analysis. Common data sources used for detecting malicious behaviour is extracted from Mitre ATTCK knowledge base and it is analysed how DRAKVUF can be used to tap into them. With DRAKVUF, one can use data sources such as system call trace, Windows registry, file system, and network connections to aid in the analysis.

Chapter 8 is used to explore the performance overhead associated with virtual machine introspection (RQ4). Five different test scenarios are devised to measure the magnitude and behaviour of performance overhead. It is clear that the performance overhead is determined by two factors: trapping frequency and average task magnitude (time spent executing monitoring code).

The magnitude of performance overhead varies between loads. The key reason behind the variation is trapping frequency and the cost of trapping a single event. For system call tracing, the number of system calls required to complete the task is fundamental. Loads that issue system calls frequently cause larger performance overhead when compared to loads that do not issue as often.

The function used to estimate performance overhead is $P(c_{avg}, f) = \frac{\Delta(t_1,t_0)}{t_0} = \frac{c_{avg} \cdot f}{1-c_{avg} \cdot f}$, where t_1 is the total execution time (fixed at 1), t_0 is the time spent executing guest code, c_{avg} is the average cost of a single trapped event, and f is the trapping frequency. The same formula can be applied for all plugins by modifying the values of c_{avg} and f.

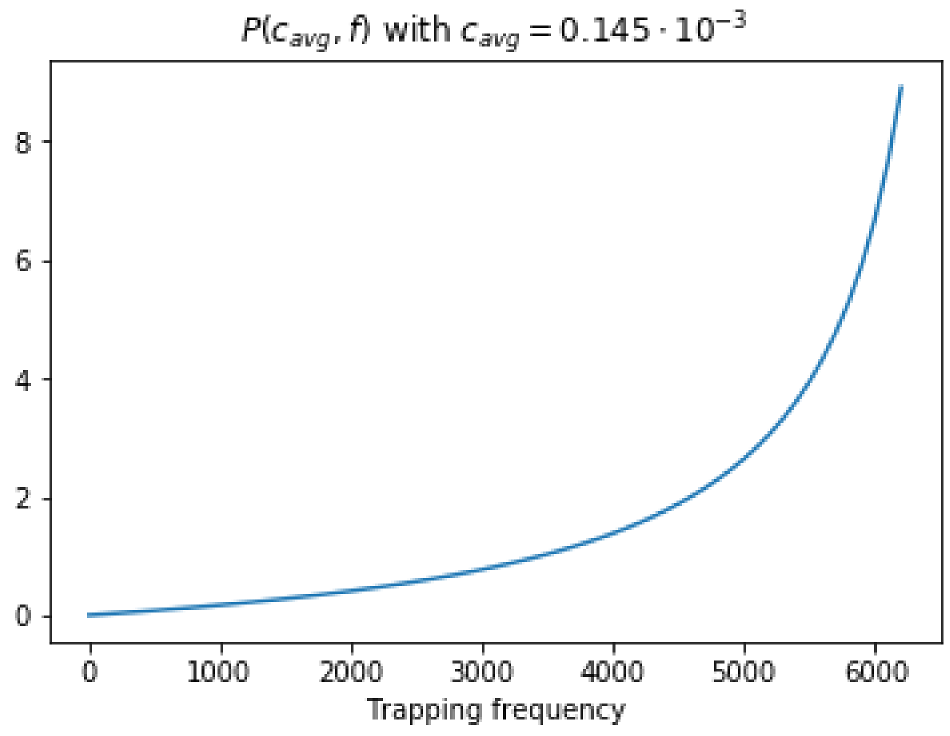


Figure 9.1: Behaviour of the performance overhead function with $c_{avg} = 0.145 \cdot 10^{-3}$. The function is plotted for $f \in (0, 6300)$.

When estimating the performance overhead of a single trapped system call, it became clear that there are differences in task magnitude between different system calls. In other words, trapping some system calls is more expensive than trapping others. One interesting research direction would be to measure the performance overhead of trapping a specific system call. More detailed knowledge about the costs of trapping different system calls would be beneficial for managing the performance overhead of the monitoring solution.

When the performance overhead of a single trapped event is known, managing the performance overhead is focused on managing the trapping frequency. As the system call frequency of a program is determined by the executed load and not by the monitoring solution, performance overhead management can be done by selecting a subset of all system calls for monitoring. If the individual trapping cost of all different system calls are known, the analyst can estimate the performance overhead for different system call distributions before deploying the monitoring solution.

The execution overhead and disk usage of file carving plugin were within acceptable boundaries if file writes do not initiate file carving. A security solution, which captures removed files could feasibly be deployed on the hypervisor without hindering the performance of the guest too much. System call monitoring and pool allocation monitoring caused performance overhead, which would hinder the performance of the guest too much to be deployed in production environments. For malware analysis purposes, the performance overhead of pool monitoring is unlikely to expose the presence of the monitoring solutions. System call monitoring is unlikely to be detected if trapping frequency stays under 1200 Hz (with $c_{avg} = 0.145 \cdot 10^{-3}$).

References

[1] R. Murali, A. Ravi, and H. Agarwal. A malware variant resistant to traditional analysis techniques. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–7, 2020.

[2] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, 2011.

[3] Y. Hebbal, S. Laniepce, and J. Menaud. Virtual machine introspection: Techniques and applications. In *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, Aug 2015.

[4] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. pages 51–62, 01 2008.

[5] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson Education Limited, 4th edition, global edition edition, 2015.

[6] Mark E. Russinovich. *Windows internals*. Microsoft Press, Redmond, 5th ed. edition, 2009.

[7] Volatility foundation, volatility wiki. https://github.com/volatilityfoundation/volatility/wiki.

[8] Linus Torvalds. sched.h. https://github.com/torvalds/linux/blob/master/include/linux/sched.h.

[9] Peter Wallin, Patrik M. Granlind, Måns Wallin, Sara Olofsson, and Richard Werner. State of cloud in the nordics - cloud maturity index 2019. Technical report, Tieto-Evry Oyj, 2019.

[10] Fatma Ahmad Bazargan, Chan Yeob Yeun, and Mohamed Jamal Zemerly. State-of-the-art of virtualization, its security threats and deployment models. 2013.

[11] Docker. www.docker.com/resources/what-container.

[12] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[13] Candid Wueest. Threats to virtual environments. Technical report, Symanytec, 2014.

[14] Brendan Dolan-Gavitt, Bryan Payne, and Wenke Lee. Leveraging forensic tools for virtual machine introspection. Technical report, Georgia Institute of Technology, 2011.

[15] 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 *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 82–91, Oct 2010.

[16] J. Pfoh, C. Schneider, and C. Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, pages 166–175, July 2010.

[17] Intel Corporation. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual - Volume 2*, 2016.

[18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection, 2003.

[19] Jonas Pfoh Mathieu Tarral, Samuel Laurén. nitro. https://github.com/KVM-VMI/nitro.

[20] Tamas Lengyel, Steve Maresca, Bryan Payne, George Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014. ACM 2014*, pages 386–395. ACM, 2014.

[21] Microsoft Documentation: Windows Debugging Tools. x64 architecture.

[22] Joni Uitto, Sampsa Rauti, Samuel Laurén, and Ville Leppänen. A survey on anti-honeypot and anti-introspection methods. In *Recent Advances in Information Systems and Technologies*, pages 125–134, Cham, 2017. Springer International Publishing.

[23] Libvmi. http://libvmi.com/.

[24] Mathieu Tarral. https://github.com/Wenzel/pyvmidbg.

[25] Lenny Zeltser. *3 Phases of Malware Analysis: Behavioral, Code, and Memory Forensics*. SANS Institute, https://www.sans.org/blog/3-phases-of-malware-analysis-behavioral-code-and-memory-forensics/.

[26] Mark Russinovich. *Windows Sysinternals*. Microsoft, https://docs.microsoft.com/en-us/sysinternals/.

[27] *Wireshark*. https://www.wireshark.org/.

[28] Raymond Canzanese, Spiros Mancoridis, and Moshe Kam. System call-based detection of malicious processes. pages 119–124, 08 2015.

[29] McAfee. *Supported IOC types*.

[30] Mitre. *ATTCK Matrix for Enterprise*. https://attack.mitre.org/.

[31] Microsoft, https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent. *IsDebuggerPresent function*.

[32] NCC group, https://github.com/nccgroup/xendbg. *xendbg - A modern Xen debugger*.

[33] FireEye, https://github.com/fireeye/rVMI. *rVMI*.

[34] I. Khan, H. Durad, and M. Alam. Data analytics layer for high-interaction honeypots. In *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 681–686, 2019.

[35] Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K. Lengyel, and K.J. Temkin. Who watches the watcher? detecting hypervisor introspection from unprivileged guests. *Digital Investigation*, 26:S98 – S106, 2018.

[36] Intel Corporation. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual - Volume 3C*, 2016.

[37] S.G. Kovalev. Reading the contents of deleted and modified files in virtualization based black-box binary analysis system drakvuf. *Proceedings of the Institute for System Programming of the RAS*, 30:109–122, 12 2018.